AN EMPIRICAL EVALUATION OF ANALYTICAL MODELS FOR PARALLEL RELATIONAL DATABASE QUERIES

THESIS

Mark C. Denham
Captain, USAF

AFIT/GCS/ENG/90-D-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/90-D-02

AN EMPIRICAL EVALUATION OF ANALYTICAL MODELS FOR PARALLEL RELATIONAL DATABASE QUERIES

THESIS

Mark C. Denham
Captain, USAF

AFIT/GCS/ENG/90-D-02

AFIT/GCS/ENG/90-D-02

AN EMPIRICAL EVALUATION OF ANALYTICAL MODELS FOR PARALLEL RELATIONAL DATABASE QUERIES

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Mark C. Denham, B.S

Captain, USAF

December, 1990

Acknowledgments

I would like to sincerely thank my advisor, Dr. Thomas Hartrum, for the guidance provided over the course of this thesis effort. I would also like to thank my committee members, Maj Mark Roth and Maj Paul Bailor. Their inputs on the first draft were very helpful as they provided many long hours of "head banging" during my final weeks at AFIT. On the serious side, they added greatly to the quality of the final paper.

I would also like to thank my fellow "parallel" students whom I have spent many hours with in room 240 over the last nine months. We have laughed together and we have been frustrated together, but most of all we have become good friends. I would like to thank my brothers and sisters in Christ for the Friday bible studies. It was a great time to get away from school work and spend some time growing spiritually.

Finally, I would like to thank my family for supporting me through the this whole ordeal. Thanks to my parents for all of the trips they made to Dayton when I was too busy to drive home for the weekend. Thanks to my sons, Christopher, Jared, and Daniel, for the hugs and smiling faces even when I was too busy to smile back. Most of all, I thank my wife, Leslie, for her love and understanding during all of those times when school interfered with our lives.

Mark C. Denham

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/90-D-02

Abstract

This ~~paper~~ documents the design and implementation of three parallel join algorithms to be used in the verification of analytical models developed by Kearns. Kearns developed a set of analytical models for a variety of relational database queries. These models serve as tools for the design of parallel relational database systems. Each of Kearns' models is classified as either single step or multiple step. The single step models reflect queries that require only one operation while the multiple step models reflect queries that require multiple operations. Three parallel join algorithms were implemented based upon Kearns' models. Two are based upon single step join models and one is based upon a multiple step join model. They are implemented on an Intel iPSC/1 parallel computer. The single step join algorithms include the parallel nested-loop join and the bucket (or hash) join. The multiple step algorithm that was implemented is a pipelined version of the bucket join. The results show that within the constraints of the test cases run, the three models are all at least accurate to within about 8.5 % and they should prove useful in the design of parallel relational database systems.

AN EMPIRICAL EVALUATION OF ANALYTICAL MODELS FOR PARALLEL RELATIONAL DATABASE QUERIES

I. Introduction

As people depend more on computers to store and manage information, the need for larger databases increases. Due to the large amounts of data that must be accessed in these large databases, future database management systems must provide faster means for accessing the data to insure that system users enjoy sustained performance levels. This is especially critical for database applications that are dependent on fast access to meet user requirements. In the late 1960s it was recognized that the conventional von Neumann architectures were not able to meet the growing non-numeric processing requirements. The conventional architectures are sequential in nature, their arithmetic logic units are designed for numeric computation, and their memory hierarchies have a "passive role in the organization (10:1)." The set related operations involved in relational database processing, on the other hand, lend themselves naturally to parallel implementations. Database systems also require frequent access to secondary storage, resulting in bottlenecks in the storage systems.

Researchers in the early 1970s began to develop specially designed architectures that would directly implement database functions in hardware and software. These implementations became known as database machines (10:1). A database machine is a dedicated backend processor whose function is to perform database queries requested by the host processor. Some of the advantages of this approach are that the database operations do not have to compete with other system functions for machine resources, specialized hardware can be used for performing the database queries, and a degree of parallelism will be inherent in the design (12:1-2).

In his PHD disseration, Kearns (12) developed a methodology for the design of parallel database machines. A set of analytical models for a variety of operations serve as the tools used with this methodology.

1.1 Problem Statement

This thesis documents the design, implementation, and evaluation of a set of parallel relational database operations in a manner similar to that proposed by Kearns (12). The objective is to show that database query performance can be improved on a commercially available parallel computer using this approach. Kearns examined the relational operators theoretically and determined the optimal methods for implementing them on a multiprocessor system. He then analytically modeled the relational operators, as well as various retrieval and update algorithms, over a range of data structures. The results of his efforts were a multiprocessor database machine architecture and a methodology to be used to design a parallel relational query processor. Kearns' work is addressed in more depth in Chapter 2.

1.2 Assumptions and Scope

Operational database management systems require a number of features in addition to the basic database operations. They require concurrency control features to control concurrent user access to data values. They also require data backup and system recovery methods to avoid loss of data in case of system failure. As the goal of this research is to measure the performance of the actual database retrieval operations, these other features will be ignored. In an operational system they would have to be considered as they will impact performance.

1.3 Approach

This research was done in four iterative phases; problem analysis, design, implementation, and evaluation. Kearns' models range from simple single step retrieval operations to complex multiple step queries in which the outputs from the retrieval operations are further

processed. The work began with the design, implementation, and evaluation of selected single step functions and then progressed to the more complex multiple step functions.

The initial problem analysis phase involved the analysis of Kearns' models, the algorithms used in them, and his proposed architecture. It resulted in the selection of two single step join operations to be implemented and evaluated.

The initial design phase involved the mapping of Kearns' single step model architecture to the Intel iPSC/1. As Kearns' architecture required a disk drive for each processor in the configuration, a disk simulator program was designed to run on specific processors on the hypercube. This phase also included the design of the initial join operations to be implemented.

From this point, the implementation, evaluation, and subsequent design phases occurred in an iterative fashion. The initial join operations and the disk simulator were first implemented on the iPSC/1 computer in the C programming language. The evaluation of these functions involved a performance analysis, based on test data, and the comparison of the results of that analysis with the performance projected by Kearns' models. The results of this analysis served as input into the design of one multiple step join operation.

The design of the multiple step join included mapping the multiple step model architecture to the iPSC/1. The selected multiple step join operation was then implemented on the iPSC/1 and was evaluated against Kearns' multiple step model for that operation.

1.4 Background

Researchers have applied a number of different approaches to the development of database machines. There have also been many different schemes developed for classifying the different implementations of database machines. For this discussion the classification scheme used by Kearns (12:6) will be used. The classification areas include conventional back-end systems, intelligent controllers, multiprocessor systems, and specialized hardware systems.

1.4.1 Conventional Back-End Systems The conventional back-end system is the simplest type of database machine to implement. It consists of a general purpose computer system running as a back-end to a host general purpose system. The host system runs the application program and passes a message to the back-end processor when a database operation must be performed. The back-end processor running a DBMS package, executes the operation and returns the result to the host processor.

According to Kearns this configuration should provide increased performance because the DBMS has a dedicated system and does not have to compete for system resources. The expected performance gain will be realized if the time saved by running the database operation in a dedicated environment is greater than the communication overhead incurred due to the message passing between the two processors (12:7-8).

1.4.2 Intelligent Controllers According to Kearns the goal of intelligent controller architectures is to improve database performance by improving the performance of data retrieval from secondary storage. By moving the data retrieval functions onto some type of intelligent controller, we can decrease the amount of data that is actually brought into the DBMS and therefore improve its performance. There are three basic types of intelligent controller architectures; processor-per-track, processor-per-head, and filters (12:9).

The processor-per-track architecture was one of the first database machine architectures proposed. It was originally proposed by Slotnick in 1970 (3:12). This design creates an associative disk by associating a processor with each head on a disk as shown in Figure 1.1. This approach allows certain search operations, such as a select, to be processed "on the fly," hence limiting the amount of data that is actually brought in from the disk (3:12). With the ability to search the entire database in one revolution of the disk, the need for structures such as indices was eliminated as a means for improving performance. RAP.2 is an example of a processor-per-track architecture (16).

This approach showed promise initially; however, it had some problems. This architecture depended on the ability to search an entire database in one revolution of the disk; however, this limited the size of the databases that could be implemented using this approach. In 1970 a single track on a disk was limited to about 15,000 bytes. With this

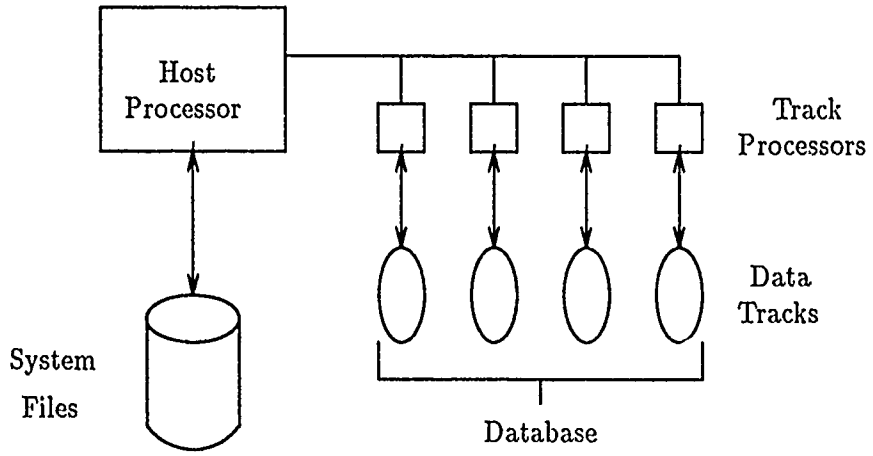


Figure 1.1. Processor-per-track Architecture (12:10)

storage limit it would have taken approximately 10,000 tracks and processors to support a medium-sized database. Because of these limits imposed by the use of disk devices, researchers also looked at the use of bubble memories and charge coupled devices (3:12). However, these technologies were never commercially viable on a large scale (3:17).

Another type of intelligent controller architecture that has been explored is the processor-per-head architecture. In this architecture a microprocessor is associated with each head of a movable head device as shown in Figure 1.2. The processor-per-head functions in much the same manner as the processor-per-track in that it reduces the amount of data that is actually passed to the DBMS. The difference is that the processor-per-head must read the data into a buffer and the processor must then perform the selection operation (12:10). In this configuration an entire cylinder of a disk can be read in one revolution. An example of a processor-per-head architecture is the DBC (1).

Performance testing performed by Dewitt has shown that the processor-per-head architecture yields very good performance for the data retrieval operations for which it was designed. He showed that with the use of a suitable index, this design performed almost as well as the processor-per-track architecture, even with significantly fewer processors (3:15). When processing complex queries such as joins, however, the processor-per track and the processor-per-head architectures both result in worse performance than conventional

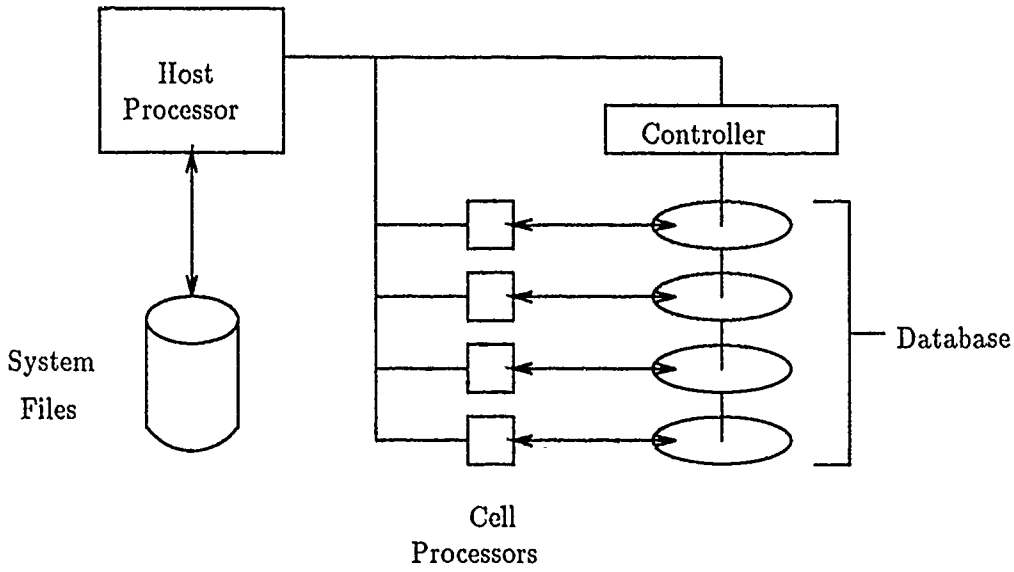


Figure 1.2. Processor-per-head Architecture (12:11)

architectures (3:15). Another drawback to this design is the need for readout disk drives. The parallel readout is required in order to read an entire cylinder at a time. With the current changes in disk technology, however, the feasibility of developing dependable parallel readout drives at a reasonable price is questionable (3:19).

The final type of intelligent controller architecture is the filter, or processor-per-disk. This architecture associates a processor with each of a number of standard disk drives as shown in Figure 1.3. All of the data from the disk is read into the processor memory and the processor performs the selection operation on the data (12:10). The selected data is then passed to the DBMS for further processing. SABRE is an example of a system that utilizes filtering (7).

Data filters, like the other intelligent controller designs, provide good performance for the basic selection type functions. They do not improve the performance of other functions such as join processing. Other techniques must be combined with the data filters to provide performance improvements over the entire spectrum of database functions.

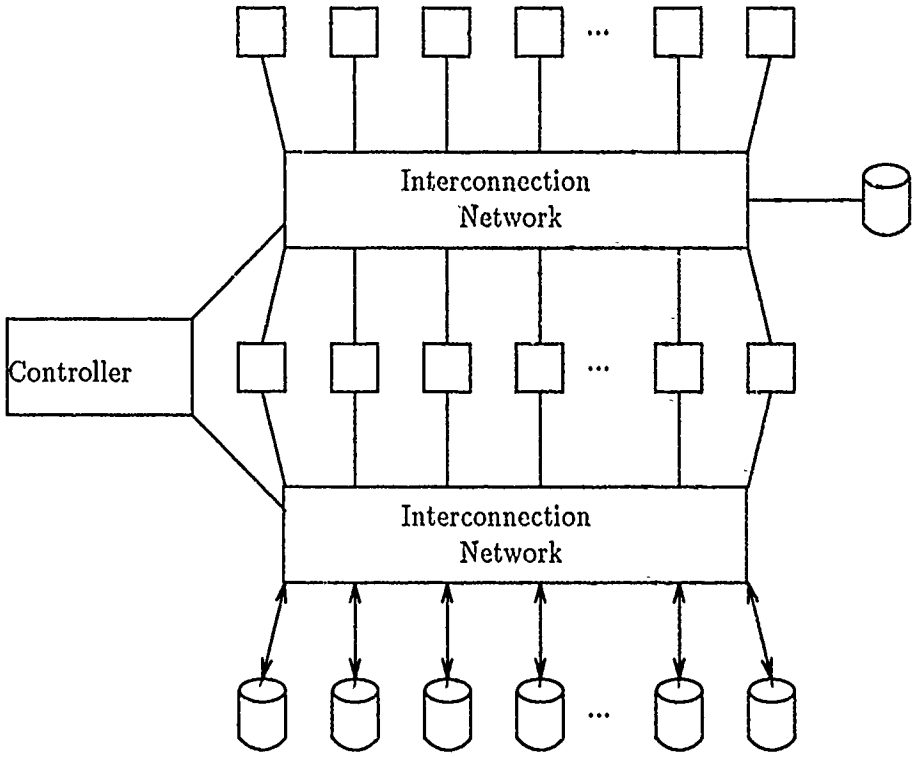


Figure 1.3. Processor-per-disk Architecture

1.4.3 Multiprocessor Systems Multiprocessor systems implement parallel processing techniques to improve database performance. There are many different approaches that can be taken to provide parallel database processing. Possible approaches range from implementing the multiple processors as data filters in a SIMD type environment, to providing a number of asynchronous processors that perform database operations in a true MIMD fashion (12:11-12). The later approach provides the possibility of implementing dataflow database architectures. In addition to the multiple processors, the multiprocessor systems also provide the capability for the processors to communicate with each other. This interprocessor communication can be provided through a bus architecture or through some type of network structure . Different interconnection schemes provide different opportunities for partitioning the data across the various processors (12:12). The remainder of this section addresses these various approaches to implementing parallel database processing.

As described in the previous section, a data filter is a type of intelligent controller

that associates a processor with each disk in a database system. All of the data is read in from the disk and a selection criteria is applied to it by the associated processor. A data filter would be implemented on a multiprocessor system by connecting each of the disks to one of the processors. The selection algorithms would then be implemented in software and synchronously run on each of the multiple processors. The "filter" processors would then send the results to other processors or to another system for further processing. In this architecture, one of the processors not being used as a filter would control the other processors. The Multibackend Database System (MDBS) is an example of a multiprocessor system that utilizes this approach (9)(19). MDBS consists of a number of query processors, each connected to a string of disk devices. The select operations are performed in these query processors. The query processors are connected, via a broadcast bus, to each other and to a backend controller (9:302-303).

The data filter approach utilized multiple processors in a synchronous SIMD manner. MIMD machines, on the other hand, consist of multiple processors that are utilized asynchronously. DIRECT is an example of a MIMD database architecture (5). The DIRECT system architecture consists of a host general purpose processor, a backend controller, a set of query processors, a set of charge coupled devices that act as a cache, a set of mass storage devices, and an interconnection matrix (crossbar switch) which ties the query processors, the cache memory, and the mass storage devices together (5:123). User queries are sent from the host computer to the DIRECT backend controller. The backend controller determines the number of query processors that are required and then is responsible for paging needed non-resident relation pages in from the mass storage devices to the cache memory (15:266-268). Multiple queries can be processed simultaneously as the different queries can be assigned to different processors at any one time. The backend controller uses two different methods for allocating the queries to the query processors. One approach calculates the optimal number of processors needed to process the query and the other uses a dataflow approach (5:128-129).

MIMD architectures, such as DIRECT, provide the opportunity to implement dataflow control in database systems. In conventional program control flow, the programmer specifies the order in which operations will occur. However, in dataflow computation the order

in which operations are executed is determined by the availability and interdependency of resources. The order of execution can be driven either by the availability of data or the demand for output (10:366). To understand the concept of data driven control in a database system, consider the idea of a query tree. In Figure 1.4, a complex relational query is

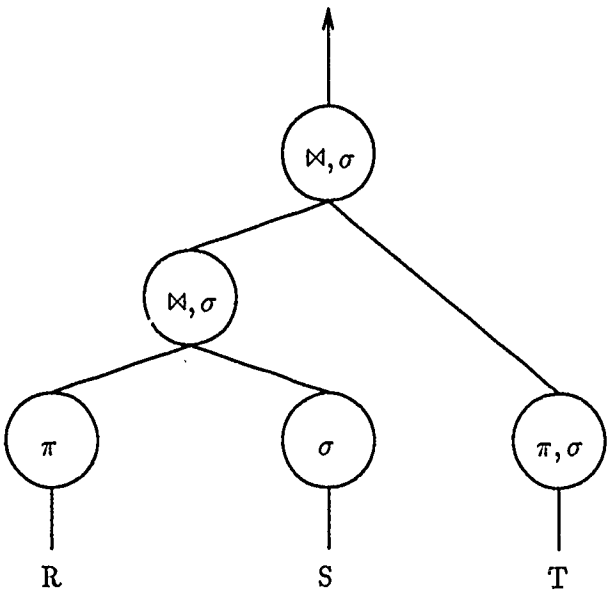


Figure 1.4. Query Tree

represented as a query tree. A query tree is composed of leaf nodes which represent the initial input relations, and inner nodes which represent operations to be perfomed on the relations. The arcs linking the nodes represent the passing of intermediate relations from the lower nodes to the higher nodes. In this example, the project operation on relation R and the select operation on relation S must both be executed before the join on the intermediate relations produced can be executed. As soon as the project and select are finished and have passed their data to the higher level node, it can be executed.

In a multiprocessor architecture, the operations in a particular node of the tree can be processed by one or more processors. On a conventional sequential computer the operations in this query would be executed serially. The project and select operations on the R, S and T relations would occur first. The join of the R and S relations would occur next. Then the final join and select would be executed. On a parallel machine, however,

the project and select operations on the R, S, and T relations would occur in parallel. Then the join and select operations on the R and S relations would be executed. Finally, the last join and select would also be performed. The performance gain would result from the various operations being run in parallel, and also from the individual operations being implemented with parallel algorithms.

Performance testing performed by Boral and Dewitt with DIRECT has shown that one of the major problems that exists with data driven computation is the overhead associated with interprocessor communication. They also state that other researchers have documented the same problem (2:373). Communication overhead is a common problem with many multiprocessor systems.

1.4.4 Special Hardware Designs Many of the database machines that have been developed by researchers have included specialized hardware to speed up database processing. According to Kearns the two main approaches that have been used include associative memories and specialized processors (12:15). Associative memory is addressed by content rather than by location. It has been applied to the retrieval of data to speed up search type operations. Specially designed processors have been implemented to perform sorting functions, join functions, and aggregate functions (12:15). Boral and Dewitt state, however, that with the gains that have been realized in processor performance, there is no need to use specialized hardware (3:21). The real bottleneck that exists in database processing is input/output bandwidth. Data can currently be processed as fast or faster than it can be read from disk.

1.4.5 Conclusion This section has documented many of the different approaches that have been taken in the development of database machines. Most of the recent research has centered on the use of multiprocessors to improve database processing performance. Two of the main problems that still exist are the lack of secondary storage input/output bandwidth and excessive interprocessor communication overhead.

1.5 Organization

The organization of this thesis follows the approach taken in accomplishing the work. Chapter II provides an overview of Kearns' work (12). It discusses the theoretical feasibility of implementing relational database operations on parallel computers, provides a summary of the model structures and model results for both the single step and multiple step models, and presents a database machine architecture proposed by Kearns. Chapter III documents the design and implementation of the single step models. Chapter IV provides a discussion of the single step implementation analysis. Chapter V presents the design, implementation, and analysis of one multiple step query.

II. Parallel Database Machine Design Methodology

In his PhD dissertation (12), Kearns documents a methodology and tools to be used to design a multiprocessor database machine that will improve the performance of database queries (12:4). He first examines theoretically the feasibility of implementing relational operators on partitioned relations. He then develops analytical models of the select, project, join, and update operations over a range of data structures and logical machine architectures. After evaluating the initial model results, he combines them to form models for evaluating multi-step query performance. He addresses the combination of multiple operators and their performance in various multi-step query approaches as well as methods for controlling resources and task allocation. Utilizing the output of these models, Kearns proposes a parallel database machine architecture. This chapter addresses each of the areas of Kearns' research.

2.1 The Feasibility of Relational Operators with Partitioned Relations

Since its introduction by Codd in 1970, the relational data model has become the most popular model to be used in the development of database systems. This is because of its logical interface and its strong mathematical foundation (10:39). In this model, data is represented in tables as shown in Figure 2.1. Each row in a table shows a relationship

Emp

name	addr	phone
Bell	4123 Lake St	539-8741
Johnson	1345 Oak St	345-3281
Jones	345 Poplar St	421-8976
Smith	2367 Main St	549-7610

Figure 2.1. Relational Table

between the values in the row. These tables in the relational data model correspond very closely to the concept of a relation in mathematics. Because of this close correspondence, tables are referred to as relations, and rows are referred to as tuples (13:45).

When implementing a problem on a parallel computer, the problem must be partitioned across the multiple processors. This can be accomplished by either assigning different functions to the available processors, or by distributing the data across the available processors. Since most database machines have experienced bottlenecks in retrieving data from secondary storage, Kearns investigates the partitioning of relations across multiple data stores to allow the parallel retrieval and processing of the relations. He proves, using the mathematical principles of relations, that relational operators are capable of executing with partitioned data (12:21).

2.1.1 Data Fragmentation There are three ways that relations can be distributed for processing. The first is storing them as complete relations, i.e not distributing them at all (12:21). The other two methods are horizontal fragmentation and vertical fragmentation. Horizontally fragmented relations are split into fragments, each of which contains a subset of complete tuples from the original relation. Horizontal fragmentation is illustrated in Figure 2.2 (12:22). The sets of tuples produced by a horizontal split should be disjoint. If they are not, duplicate tuples will have been introduced into the relation.

Emp1

name	addr	phone
Bell	4123 Lake St	539-8741
Johnson	1345 Oak St	345-3281

Emp2

name	addr	phone
Jones	345 Poplar St	421-8976
Smith	2367 Main St	549-7610

Figure 2.2. Horizontal Fragmentation

Vertical fragmentation, on the other hand, splits the relation between attributes. Vertical fragmentation is illustrated in Figure 2.3 (12:23).

2.1.2 Conclusions Kearns' results show that most relational operators are capable of processing fragmented data. The feasibility for each of the considered operations to process horizontally and vertically fragmented relations are shown in Table 2.1 (12:48).

As stated above, one of the primary motivations for considering fragmented relations is to allow the data to be distributed across multiple storage devices to be retrieved and processed concurrently. As illustrated in Table 2.1, only half of the considered operators

Emp Addr

name	addr
Bell	4123 Lake St
Johnson	1345 Oak St
Jones	345 Poplar St
Smith	2367 Main St

Emp Phone

name	phone
Bell	539-8741
Smith	549-7610
Johnson	345-3281
Jones	421-8976

Figure 2.3. Vertical Fragmentation

Table 2.1. Feasiblity of Relational Operators with Fragments

Operator	Horizontal	Vertical
Select	Yes	Yes
Project	Yes	No
Join	Yes	Yes
Product	Yes	Yes
Union	Yes	No
Difference	No	No

can process vertically fragmented data and produce the desired result. The horizontal fragmentation of data, on the other hand, shows much more potential. All of the considered operators, except for difference, can produce the desired results from horizontally fragmented data. As an example, we can examine the select operator and see that if a relation is distributed across n disks which are connected to n different processors, a speedup of n can theoretically be obtained by concurrently reading the fragments from the disks and then performing the select operations on the n fragments in parallel.

Kearns concludes that horizontal partitioning of the relations shows greater promise for improving the performance of the relational operators. He therefore selects horizontal partitioning as the method for the remainder of his work (12:49).

2.2 Single Step Query Models

When comparing database systems, there are many criteria that can be used. These criteria may include the number of features or the type of user interface provided. However, the most used, and possibly most important comparison, is the performance provided by

the systems (12:50). The performance time to process a relational database query is composed of the time to compile the query, the time to retrieve the desired data through the implementation of the relational operators, and the time to send the data to the desired location. Assuming that the result will be sent to a local user and not transferred over a slow LAN, retrieval time contributes the largest percentage to the overall performance time (12:50).

Data retrieval is performed by reading data from some secondary storage device, such as a disk, and evaluating it using the relational operators to determine if it meets the retrieval criteria. The performance of a retrieval operation depends on the structure used to store the data on the storage device, the algorithm used to read the data, the amount of data that must be retrieved, the number of storage devices used to store the data, the number of processors used to retrieve and process the data, the performance characteristics of the hardware devices, and the number of users on the system (12:50).

2.2.1 Model Structure Kearns develops analytical models to address data retrieval performance and also to address database update performance. The purpose of these models is "to determine the effects and performance of various algorithms with different storage structures (12:51)." The models consist of performance time equations expressed in terms of hardware performance parameters. These hardware parameters can then be changed to account for different hardware devices. The relational operators that Kearns actually models are the select, project, and join. For each of these he models the performance time for various architectures, data storage structures, and algorithms.

2.2.1.1 Machine Architectures Kearns' models evaluate the various relational operators over four different machine architectures. They are single processor-single disk, single processor-multiple disk, multiple processor-single disk, and multiple processor-multiple disk (12:51). For the multiple processor cases he varies the number of processors to evaluate the effect on runtime. He makes the assumptions that there is no disk contention in the single disk environments and that the multiple processors can communicate directly in the multiple processor environments. Therefore the models do not include any time delay for disk contention or processor communication (12:51).

2.2.1.2 Data Storage Structures The relations in a relational database system are stored in the form of tables. These tables can then be stored on disk in a variety of manners. They can be sorted and stored in some logical order, or they can be stored at random. Therefore a relation can be referred to as ordered or unordered. A relation can also have one or more associated indices with entries that point to particular locations in a relation. For a data storage structure to be considered indexed for a particular operation, it must contain an index on the attribute of the relation that is being evaluated for that operation.

For the single disk cases, Kearns examines four data storage structures. They are unordered, unindexed; unordered, indexed; ordered, unindexed; and ordered, indexed (12:52). For the multiple disk cases, there are other aspects of the data storage structure that must be addressed. The first is fragmentation method. As stated earlier Kearns assumes horizontal fragmentation based on his analysis of the feasibilty of relational operators with fragmented relations. The fragmented relations can also be ordered or unordered. Another aspect of fragmented relations is the distribution method used to partition the relation across the multiple disks. The three primary methods are round robin, ordered distribution, and hashing. The round robin method evenly distributes the tuples in an unordered manner. The ordered distribution method evenly distributes the tuples in an ordered manner. This method may require extra reads and writes from the disks to maintain the sorted order. The hashing method involves hashing the tuples to particular disks based on static boundries. This method will result in an ordered relation; however, it may not be uniformly distributed across the disks (12:53). The final aspect that can be considered for the multiple disk cases is indexing. With fragmented relations, local indices can exist for each disk or a global index may be maintained for the entire relation (12:54).

2.2.2 Model Results Kearns concludes that the model results cannot be used to prove definitively the algorithm and storage structure that is "best" for a given application. They can, however, point to the expected results with certain workload parameters(12:199). As it would require the implementation and evaluation of more than 200 models to prove them correct, Kearns opted instead to compare his results with those of Hawthorn and

DeWitt (8). Their results have been validated through benchmarking. The following sections summarize the model results for the select, project, join, and update operations.

2.2.2.1 Select The basic function of the select operator is to retrieve tuples that meet a specific selection criteria from a relation. There are two basic categories of selection function that Kearns modeled. The first occurs when only a few tuples meet the selection criteria (12:200). An example of this type of select operation is a selection based on a specific social security number from a company's personnel database. Because social security number is a unique key, only one tuple will be selected. The other category of the select operation is the many tuple selection (12:200). An example is the selection of employees, from the same personnel database, based on the department in which they work. Assuming that many people work in each department, a large number of tuples will be selected. Kearns classifies his select models on these two categories. His model results show that the performance of the select operation, for both the few tuple and many tuple cases, is highly dependent upon the data structure used. He concludes that the best data structure to be used for the select must be determined by the designer based upon the specific workload and the impact of the data structures on the other database operations (12:202).

2.2.2.2 Project The project operator creates a result relation that is composed of a subset of the attributes in the original relation. Since the result relation will contain a portion of each of the tuples from the input relation, each tuple in the input relation must be accessed. Therefore, the projection operation is independent of the input relation data structure (12:203). The model results show that the main performance impact is the removal of duplicate tuples that may be produced from the projection operation when the key attribute is eliminated. When this occurs, some method for comparing the tuples to identify and eliminate the duplicates is required. This may require a sort merge operation in a multiprocessor environment (12:100). The performance is greatly decreased when duplicate removal is introduced.

2.2.2.3 Join The join operation requires that tuples from two input relations be compared and joined to produce a new result relation tuple if the join criteria are met. Kearns' models addressed only the equi-join operation. The most straight forward approach is to compare each tuple of one relation with each tuple of the other relation. This is known as the nested-loop join algorithm and requires $n * m$ compare operations. The number of compares can be significantly reduced through the use of algorithms which group the tuples of each relation based upon the join attribute. The join operation can then be performed by comparing fewer tuples. Two algorithms of this type that Kearns modeled are the sort-merge join and the bucket join (12:205,208).

The model results show that parallel processing can be used to improve the performance of all of the join algorithms. It is especially effective for the grouping algorithms as each processor can perform the join operation on a specific group of tuples in parallel with the other processors. The bucket join is potentially the fastest algorithm. It achieves its best performance when the relation fragment sizes are constant across all processors (12:206,207).

Kearns concludes that the join operation is sensitive to the data structure used (12:205). To implement the grouping algorithms, the relations must be sorted on the join attribute and must be spread evenly among the processors. This implies that the relations must either be stored in sorted order and spread uniformly across the disks, or they must be sorted and redistributed across the processors before the join operation is performed. The results show that the most efficient approach in a parallel environment is to redistribute the relations and then perform a bucket join algorithm. When implemented in this manner the join operation performance is not dependent upon the actual data storage structure used on disk (12:220). Relation indices were found to actually decrease the performance of these algorithms (12:205).

2.2.2.4 Update The update operations modeled were insertion, modification, and deletion. These are the only operations Kearns modeled that do not actually retrieve data from the database and send it to the user. They instead retrieve data from the database, alter it, and rewrite it to the database (12:215). The assumption was made that

Table 2.2. Summary of Kearns' Results (12:222)

OPERATION	BEST CASE	GENERAL CASE
Select (few tuples)	Indexed-Unordered	Unindexed-Unordered
Select (many tuples)	Indexed-Ordered	Unindexed-Unordered
Project	N/A	N/A
Insert	Indexed-Unordered	Unindexed-Unordered
Delete	Indexed-Unordered	Unindexed-Unordered
Modify	Indexed-Unordered	Unindexed-Unordered
Join	Bucket-Join	Undetermined
Product	Nested-Loop	Nested-Loop

all updates are required to be done immediately. The alternative to immediate updates is that the multiple updates could be collected and performed at one time in a batch operation. If the updates are performed in a batch operation, the requirements are similar to those of the select operation. Otherwise the best data structure is the unordered, unindexed which provides acceptable performance for a mixed environment. The size of this type of data structure can also be increased by adding more disks without requiring any reorganization of the existing data (12:220).

2.2.2.5 Conclusions The model results show that the use of multiple disks to allow parallel data retrieval is definitely desirable for all operations. In many cases the specific data storage structure to be used is dependent upon the specific user requirements. Through the use of this methodology, a designer should be able to identify the desired data storage structures and algorithms to best suit his specific requirements. Table 2.2 shows the data storage structure or algorithm that produces the best case and general case performance based on Kearns' particular model inputs. The best case algorithm or data storage structure refers to the algorithm or data storage structure that provides the best performance for a particular best case operation scenario. The general case algorithm or data storage structure is the algorithm or data storage structure that provides the best performance over a range of operation scenarios.

2.3 *Multiple Step Queries*

The models described to this point have dealt with single operation or single step queries. Most queries performed in an operational relational database system, however, require multiple relational operations to be performed to produce the desired result. Figure 2.4 represents one such query in the form of a binary tree. The leaves of the tree represent

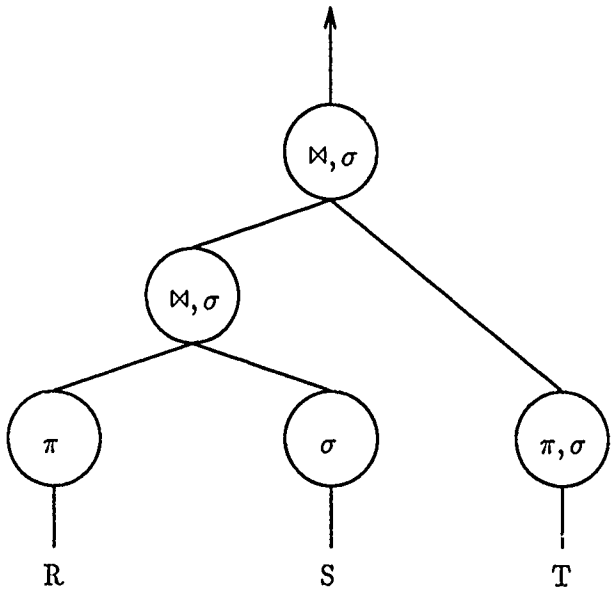


Figure 2.4. Query Tree

the initial input relations and the inner nodes represent the operations to be performed on the data. Progressing up the tree, each arc represents the passing of an intermediate relation from a lower level operation to the next level operation. The final result of the query is produced by the root node of the tree.

The primary multiple step query issues that Kearns addresses are process allocation, data distribution, and processor interconnection (12:239). He assumes a fully connected system which simplifies the processor interconnection issue. The actual queries that he models include select, project, and join operations. For the join operations he utilizes the bucket join algorithm as it proved to be the most efficient in the initial models.

2.3.1 Combined Operators A large percentage of the time required to perform multiple step queries is spent storing and retrieving intermediate relations to and from disk (12:228). To reduce the size of these intermediate relations and to reduce the number of operation nodes in the query tree, Kearns investigates the combining of the relational operators. These combined functions would perform multiple operations on the same data while it is in memory. An example is the combination of the select and the project operations. If the operations are not combined, a relation would have to first be read into memory, the select operation performed on it, and the result written to disk. The result of the selection would then be read back into memory for the project operation to be performed. If the operations were combined, the initial relation could be read in just once. In one pass over the data, the tuples that meet the selection criteria could be selected and any attributes not specified for the project operation discarded before the tuple is inserted into the result relation. If duplicate removal is required for the project operation, a second pass over the result tuples may also be required.

Kearns' model results show that the select and project operation performance can be improved significantly through the implementation of a combined *sel-proj* operation (12:228). The sel-proj combined operation requires just one pass over the input relation to perform both the select and the project operations. He also concluded that the sel-proj operation can also be combined with any binary operation, such as the join operation (12:230).

The result of combining operations is a *normal form query tree* that contains two types of nodes. The bottom level nodes are the *retrieval nodes.* They retrieve relations from disks and perform only sel-proj operations. The higher level nodes are the *processing nodes* (12:231). They perform join operations on two input relations. In the case of the bucket join algorithm processing, which is the algorithm that was modeled, the processing nodes perform hash and join functions.

2.3.2 Process Allocation Process allocation can be either static or dynamic. Static allocation entails assigning the available processors to specific tasks when a query is compiled. This approach allows for data-flow control of the query. Data-flow processing is

where the order of execution is determined by the availability of data (12:270). In the case of a join operation that recieves its input relations from two select operations, the join operation will be performed when the processors assigned to it receive the input relations from the select processors.

Dynamic allocation, on the other hand, dynamically allocates tasks to the processors as the query is executed. The purpose of the dynamic method is to balance the work across the available resources throughout to entire execution of the query. The dynamic allocation process may be controlled by a central controller, which may tend to become a bottleneck, or through some form of distributed control function.

In addition to pure static or dynamic allocation, Kearns also addresses a hybrid control scheme in which the available processors are partitioned into groups which perform specific operations; however, the work is balanced across the processors within a particular partition (12:270-271). This scheme is able to balance the work across the available resources to some extent, without the bottleneck of one centralized controller.

2.3.3 Data Distribution In his models Kearns evaluates two schemes for storing the input relations on the disks. The first scheme spreads both input relations across all available disks. The second spreads each relation across half of the available disks. He found that if the sizes of the input relations are about the same, the schemes will result in similar performance. However, if one of the relations is significantly larger than the other, the first scheme will outperform the second (12:265).

2.3.4 Conclusions The model results show that performance is improved by implementing combined operators to reduce the size of the intermediate relations. They also showed that assigning groups of processors to perform specific operations and then spreading the processing load among the processors in the group provides some of the benefits from dynamically spreading the work among processors but avoids the bo tleneck of a centralized controller. Kearns concludes that the existence of two types of ;ical operations to be performed, initial retrieval and binary operations, results in a lc$_{\epsilon}$ mapping to a two stage query processor architecture (12:271).

2.4 *Database Machine Architecture*

Kearns' final chapter presents a logical database machine architecture, based upon the analytical model analysis documented previously. This architecture is not presented as the perfect machine, but is presented and discussed to illustrate the use of the models in the systematic design of a database machine architecture.

The architecture that Kearns proposes is a backend processor that recieves optimized queries from a host processor, performs the query processing, and either returns the result to the host processor, or stores the result on disk. As discussed in the previous chapter, the processor nodes in the multiple step model can be divided into two types, retrieval nodes which read and write relations from disk and perform sel-proj operations and processing nodes which perform binary operations such as joins (12:274). Therefore, each node in the database machine architecture is either a part of the retrieval layer or the processing layer as shown in Figure 2.5. The following sections summarize the retrieval and the processing node layers. The final section discusses considerations for mapping the architecture to a physical machine.

2.4.1 Retrieval Layer The function of the retrieval layer is to perform data retrieval and update operations on the data residing on disk. The retrieval function involves reading data from the disk, filtering it, and passing it to the binary layer nodes for further processing (12:278). The filtering process entails performing the combined sel-proj operation to reduce the volume of data passed to the binary layer. The update functions simply involve writing the updated relations back to the disk. The functionality of the retrieval layer nodes is intentionally limited to aid in reducing the I/O bottleneck to the disks. By not utilizing these nodes in performing the binary operations, they are free to perform I/O operations while the processing nodes are performing more complex operations (12:278). The decision was also made to let the processing layer nodes handle their own intermediate relation storage rather than passing the data to the retrieval layer. This avoids the reverse flow of data from the binary layer to the initial layer and also aids in I/O bottleneck reduction (12:279).

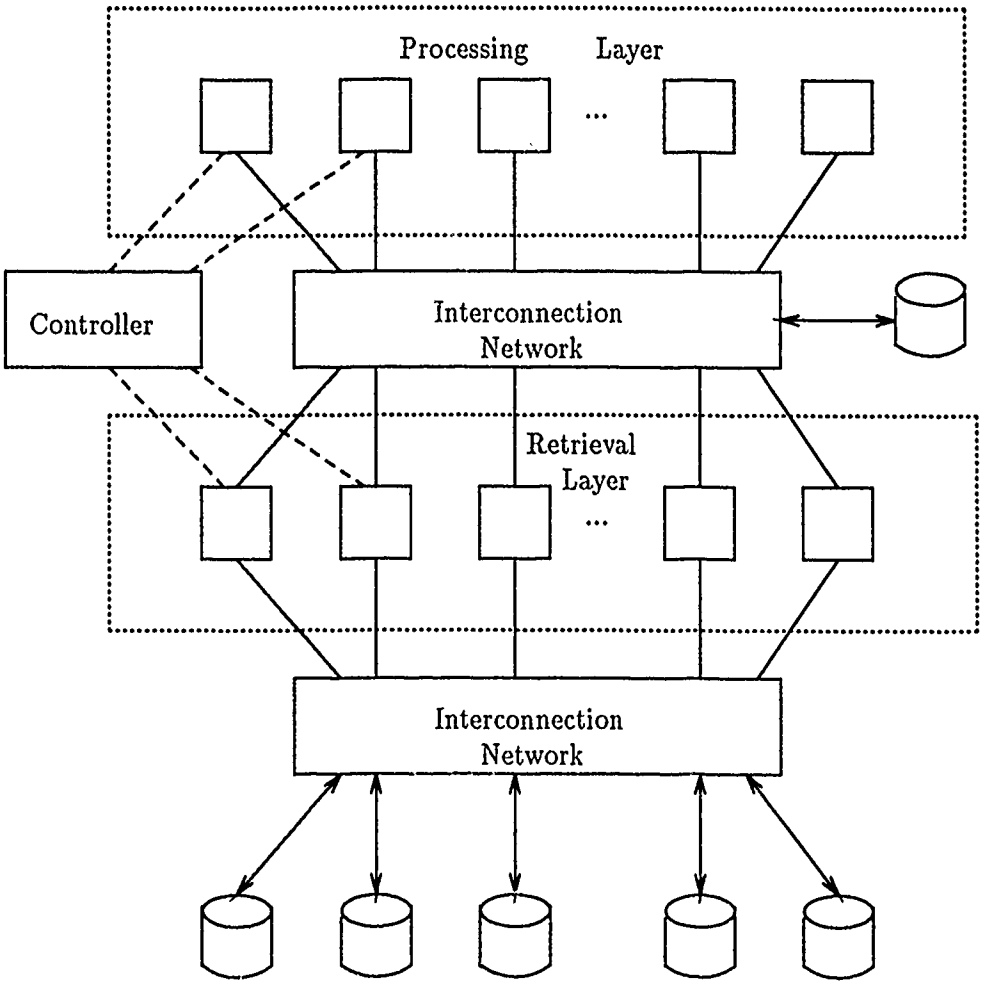


Figure 2.5. Kearns' Logical Architecture (12:282)

2.4.2 Processing Layer The function of the processing layer is to perform the binary operations required to perform a database query. The binary operations are those such as join, that require more than the sel-proj processing at the retrieval layer (12:280). The processors at this layer perform either bucket or nested loop operations. The bucket operations are performed by grouping the input tuples into disjoint sets, or buckets, and then distributing the buckets among the available processors. The processors then sort the buckets and perform the required binary operation. If the buckets are larger than the available memory of the processors, some type of secondary storage is required to store portions of the buckets (12:280). The result of the retrieval is then passed to the appropriate location.

2.4.3 Physical Mapping The final step of the database machine design process is the mapping of the high level logical design to an actual physical architecture (12:283). As the various functions to be performed by the database machine will probably not perform optimally in the same processing environment, design decisions must be made that may result in optimal performance for some functions, and less that optimal performance for others. The user requirements should drive the performance priorities given to the various functions. This is the point at which Kearns incorporates the use of his models. The models provide a means to vary the performance of system parameters in order to determine the impact on the performance of the different functions.

Figure 2.6 shows a possible physical architecture proposed by Kearns. It consists of the processing layer processors, the retrieval layer processors, and a back-end controller. Disks are located at both the retrieval layer and the processing layer. The disks at the retrieval layer are used to store the database relations and the disks at the processing stage are used to store intermediate relations during binary query operation processing. He also shows an interface memory to facilitate communication between the retrieval and processing layers.

The function of the retrieval layer processors is to read input relations from disk and perform the sel-proj combined operation. Kearns used the model results to conclude that at the retrieval layer,

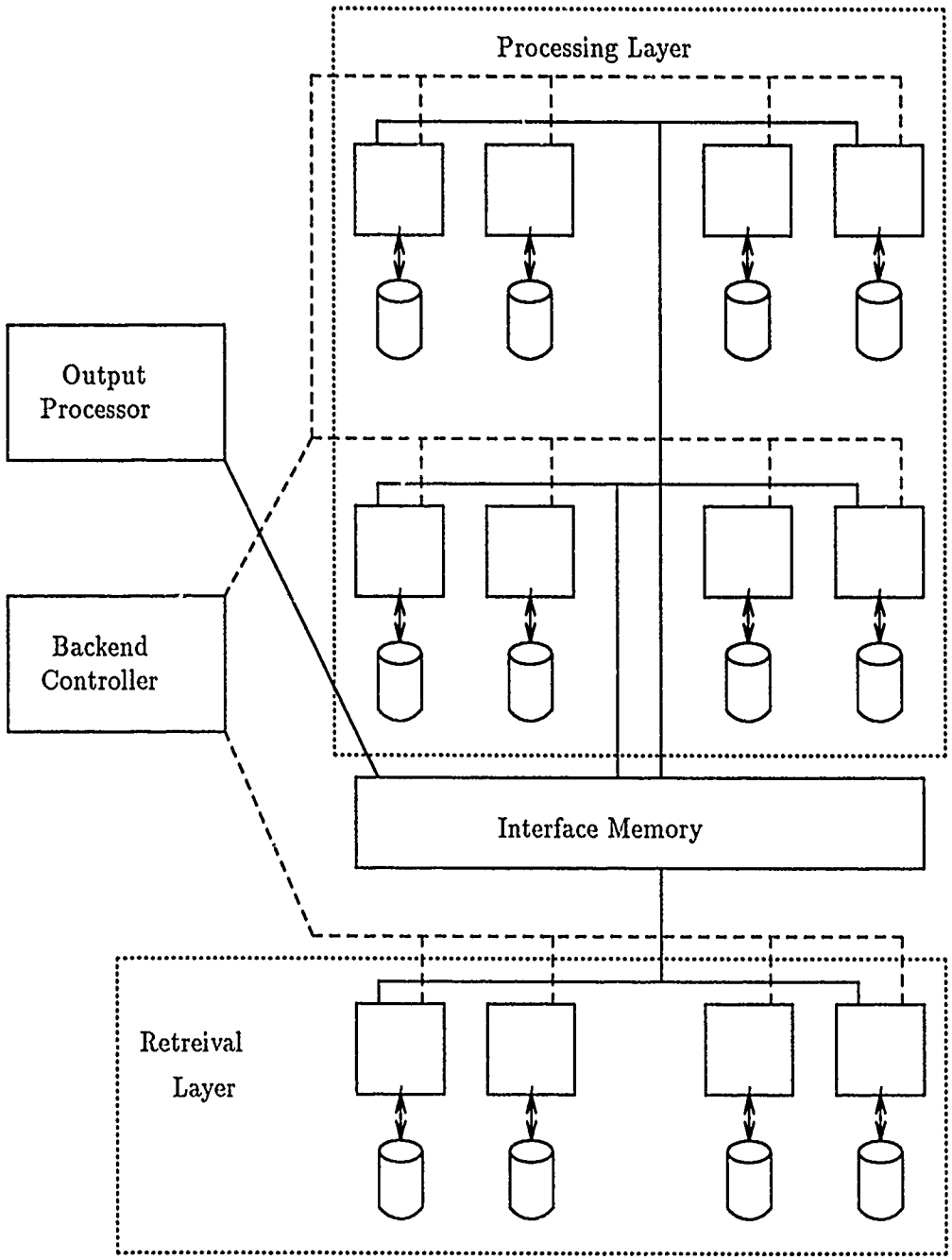


Figure 2.6. Kearns' Physical Architecture (12:284)

- all relations should be evenly distributed across all disks.
- a single processor should be associated with each disk.
- certain data structures could provide optimized retrievals for certain operations.
- optimized retrieval performance could also be obtained utilizing an unordered-unindexed data structure with additional hardware (12:285).

After performing the sel-proj operation, the retrieval layer processors may pass their results to the processing layer for further processing. It is obvious that a fully connected communication interface between the layers would provide the best performance. Kearns includes a shared memory as the interface between the layers. Because this memory can be read by all processors, it is logically a fully connected interface. It also provides the capability for broadcast communication as all processors can read the same memory location (12:287).

The processing layer consists of the processors that are to perform the binary operations and disks that are used to store intermediate relations. In Kearns' architecture a disk is associated with each processor. Another configuration considered associates a disk with a group, or partition, of processors. This would, however, degrade performance as there would be contention for the disk between the multiple processors in a partition (12:290).

The final two components of the architecture are the controller and the output processor. The controller is a processor that serves as the input interface between the host computer and the database machine. The controller is responsible for receiving queries from the host and then assigning the tasks to the database machine processors (12:291). The output processor, on the other hand, serves as the output interface to the host processor. When there is a requirement to provide query result relations to the host processor, this is the responsibility of the output processor. This may include the merging of fragments of the result relation as they are received from the processing or retrieval layer processors.

2.5 Summary

This chapter has discussed Kearns' methodology for the design of a parallel database machine. The initial section discussed the feasibility of performing relational database operations on distributed data. This capability is essential if the relational query processing is to be parallelized. Next, a summary of Kearns' single step and multiple step query models was provided. These models comprise the tools to be used in the design of the database machine. The final section presented Kearns' proposed parallel database machine architecture. This architecture was proposed to demonstrate some of the aspects of the system that can be addressed with the models.

In his analysis, Kearns spent the most time discussing the different join processing methods. This is because join processing is the most resource intensive operation performed by a relational database system and hence provides the greatest potential for performance improvement. Based on these facts, the implementation portion of this thesis will also concentrate primarily on join processing.

III. Single Step Query Model Design and Implementation

3.1 Introduction

As stated in Chapter 1, the purpose of this thesis is to design, implement, and evaluate parallel relational database operations based on the analytical models developed by Kearns (12). His models are divided into single step models and the more complex multiple step models. This chapter describes the overall design and the implementation of the single step query models. As stated in Chapter 2 this implementation concentrates on equi-join processing.

Kearns' single step query models are based upon a hardware architecture in which each of the processors, in a multiple processor system, is associated with a secondary storage device such as a disk. The database relations are then assumed to be horizontally partitioned and spread uniformly across the available disks. This allows for the relation fragments to be read in parallel, reducing the I/O bottleneck. The processors are assumed to be able to communicate either directly, or indirectly, with any other processor. The system also has a controller that receives an optimized query from the host processor and broadcasts the required operation to the processors. The query result relation may either be stored to disk or merged and returned to the host processor. Such an architecture is shown in Figure 3.1.

3.2 iPSC/1 Hypercube Architecture

The machine selected to host this implementation was the Intel iPSC/1 Hypercube. The iPSC/1 is a multiple instruction, multiple data (MIMD) computer. It is composed of the cube and the cube manager. The cube used for this project consists of 32 identical cube nodes which are connected via high speed channels to form a parallel computer. Each node consists of an Intel 80286 microprocessor and associated memory. The nodes communicate with each other by sending messages over the high speed channels. There is no shared memory between the cubes. The cube manager is also an Intel 80286 microprocessor. It serves as the user interface to the nodes in the cube. The cube manager is the only processor

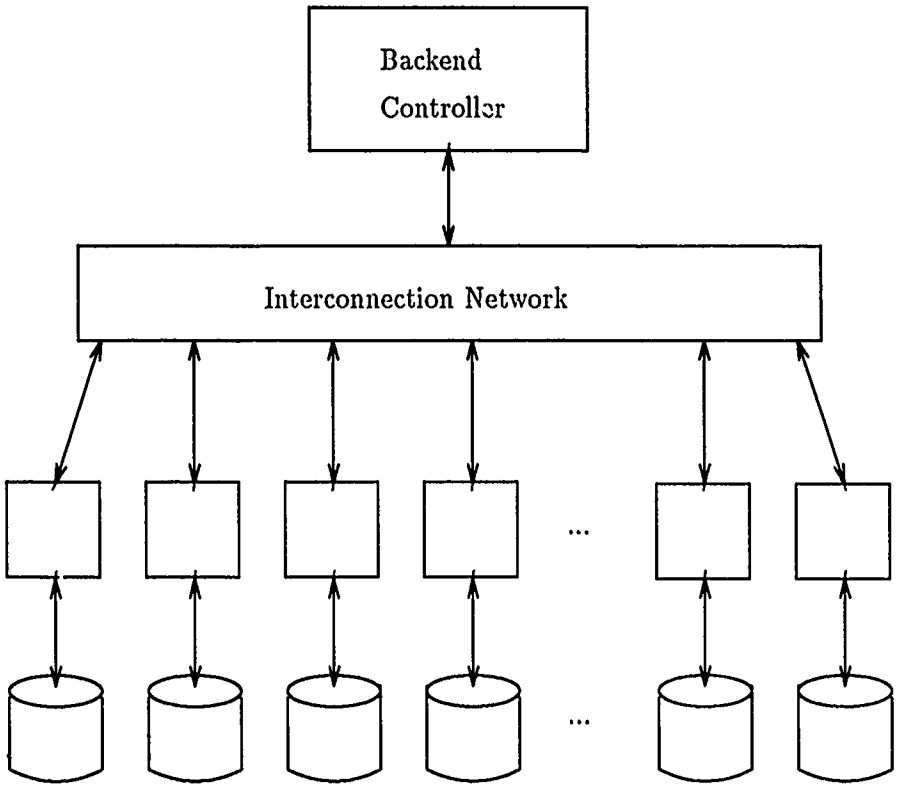


Figure 3.1. Single Step Query Architecture

in the iPSC/1 that has associated secondary storage. The cube manager communicates with the cube nodes via a standard Ethernet connection (11:2-1).

The nodes in the iPSC/1 cube are arranged in a "hypercube" interconnection topology. A 32 node hypercube is said to have 5 dimensions as it has 2^5 nodes. The hypercube nodes are not fully connected as each node cannot communicate directly with every other node in the cube. However, the nodes can all communicate with each other indirectly by passing messages through intermediate nodes. Nodes that are directly connected are said to be nearest neighbors. Each node in a dimension d cube has d nearest neighbors (11:2-3). Figure 3.2 illustrates a dimension 3 hypercube. It has 2^3, or eight, nodes, and each node has three nearest neighbors. We can use this cube to illustrate how other communication topologies are implemented on a cube connected computer. Figure 3.3 (a) shows the cube configured into a ring structure and Figure 3.3 (b) shows it configured into a tree topology.

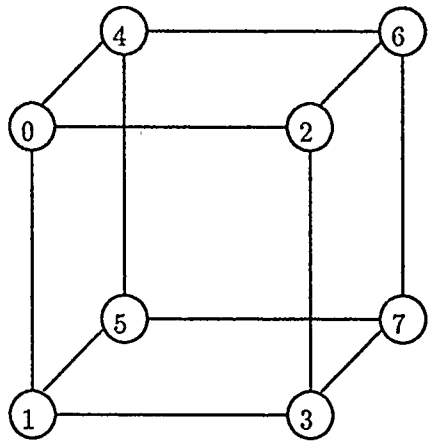
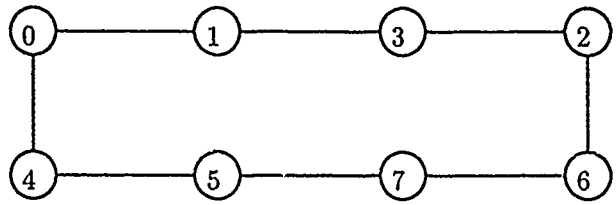


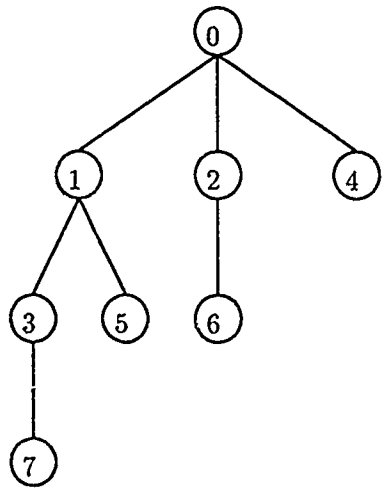
Figure 3.2. Dimension 3 Hypercube

3.3 Physical Mapping

For the most part, the architecture assumed by Kearns in his single query step models maps nicely onto the iPSC/1 architecture. The cube manager serves as the database machine controller and hence is responsible for initializing the operations on the processor nodes and merging the query results as they are received from the nodes. The cube nodes are not fully connected physically through shared memory. However, they can all communicate indirectly through intermediate nodes. The only inconsistency with the iPSC/1 architecture is that the nodes in the cube do not have direct access to secondary storage devices. Kearns' models assumed that each processor was connected to a unique disk to avoid disk contention and to allow concurrent disk access by all of the processors. This problem is solved by designating 16 of the cube nodes as disk nodes and 16 as processor nodes. Each of the disk nodes runs a disk simulation process and is associated with one of the processor nodes. The processor nodes read from and write to the disk nodes by passing messages. One of the advantages of the simulated disks is that they allow the disk access times to be varied to investigate the impact on the entire query performance.



(a) Ring



(b) Tree

Figure 3.3. Ring and Tree Topologies

3.3.1 Processor Nodes The function of the processor nodes is to read in blo . of the input relations from the disk nodes and perform a specified parallel relational query. Each of the processor nodes is associated with one disk node which is one of its nearest neighbors. Several system attributes can be varied in order to investigate their effect on query performance. The variable processor node attributes are the number of processors and the processor memory size. The number of processors can vary from 4 to 16. Although the physical memory resident on the nodes of the iPSC/1 is large enough to hold both input relations of the test database, the actual amount of memory used will be limited to allow investigation of the performance impact of varying memory sizes. The memory size can

be specified as a multiple of the input blocksize. The hypercube topology of the iPSC/1 allows the nodes to communicate in various ways, depending on the particular parallel algorithm being implemented. A different process is loaded onto the processor nodes to perform each operation. The specific algorithms implemented on the processor nodes are discussed later.

3.3.2 Disk Nodes The function of the disk nodes is to generate input data for the operations to be performed and to provide a simulated secondary storage capability to the processor nodes. They provide the processor nodes the capability to read fixed blocks of tuples from the generated relations and also to store and retrieve intermediate relations as needed during query execution. The response time for disk node operations can be varied based upon specified disk performance characteristics. The number of disk nodes may vary from 4 to 16 and is always equal to the number of processor nodes.

3.3.2.1 Data Generation When initialized, the disk nodes automatically generate data for two input relations which are horizontally partitioned and spread uniformly over the specified number of disk nodes in an unordered fashion. This simulates a round robin method of placing the tuples into the relations. The number of blocks in the relations and the blocksize can be varied, but each "disk" contains the same number of blocks. Possible ranges are from a 16 block relation (1 per disk) to a max of 1536 (96 per disk).

The tuples in each of the relations have a simple schema consisting of two integer fields, with one specified as the join field for equi-join processing and the other being a random integer. While this simple schema serves the purposes of this research, more elaborate general purpose test database schemas have been developed as demonstrated by Strawser (17). It may be noted that the relations do not have a unique key field. A unique key is required for relations in operational relational database systems, however, this is not an operational system. The tests run for this work do not require a unique key field. If a unique key field is required, the random integer field can be assigned a unique integer for each tuple.

The domain of the join field is the set of positive integers from 0 to some specified

range value. The assignment of join field values to the tuples in the relations is a three step process. The first step is to established the size of the desired result relation produced when the two input relations are joined. This size is calculated using the desired join selectivity factor (jsf). The jsf specifies the size of the result relation as a fraction of the size of the cross product of the input relations (12:119). So, given two input relations with 10 tuples each and a jsf of .1, the cross product will contain 100 tuples and the join result will contain 10 tuples. Once the desired size of the result relation is established, the next step is to determine the the number of common join values that will satisfy the equi-join criteria and place them in the relations. Eight common join field values are identified that are uniformly distributed across the join field domain. These eight common join values are repeatedly assigned across the relations at uniform intervals until the desired number of common join values have been assigned to each relation. This results in an unordered, uniform distribution of the join values across each relation. The final step is to assign join field values to the tuples that have not been assigned the common join values. These tuples receive a uniform distribution of the rest of the values from the finite integer domain in an unordered fashion. To avoid inadvertent joins, all of the non-common join values for one of the relations are even, and for the other relation they are odd. The actual number of result tuples produced from this data may only be an approximation of the specified jsf. Appendix E contains further discussion of the actual result relation sizes produced different values of the jsf.

The disk nodes allow the processor nodes to read the generated relations in multiples of the generated blocksize. A processor node requests a read operation by sending a read message to a disk node. The read message specifies the relation to be read from, if the read is to begin from the current location of the read pointer or from the beginning of the relation, and the number of blocks to be read. The processor node then waits for a return message from the disk node. The disk node returns one message containing the specified number of blocks to the processor node.

3.3.2.2 Temporary Relation Storage In addition to generating the initial input relations, the disk nodes also provide the processor nodes with the capability to tem-

porarily store intermediate relations during query processing. For the single step models, the schema of the intermediate relations will be the same as for the initial relations. The processor nodes write to the disk nodes by sending write messages which may contain either one or multiple blocks of tuples to be stored on the disk nodes. The write message type specifies the particular relation the tuples are to be stored in. The receiving disk node then blocks the written tuples into as many complete blocks as possible and places remaining tuples into an incomplete block. When stored, an incomplete block requires the same amount of space as a complete block. It contains an end of block pointer after the last tuple. The intermediate relations may be read from the disk nodes in the same manner that the initial relations are read. The read message will specify the particular intermediate relation rather than an initial relation to read from. The data blocks returned for an intermediate relation may or may not be complete, depending on how they were written to the disk nodes.

3.3.3 Disk Node Response Time Kearns' models include three disk performance parameters that can be varied to evaluate query performance with different types of disks. These are the average disk access time (seek and rotational delay), the track-to-track seek time for one track on the disk, and the time to transfer one block of data to or from the disk. The disk nodes provide the capability to vary these performance characteristics in order to simulate the response times for various types of disks. To change the performance characteristics of a disk node the disk simulation code must be recompiled. When a read or write operation is requested, the disk node calculates the proper delay time based upon the performance characteristics, the number of blocks of data that are being transferred, and the required transfer time between the disk node and the processor node. It then invokes a delay routine which provides the required delay.

3.4 Join Algorithms

A number of different sequential algorithms have been developed for performing relational joins. Three of the most common are the nested-loop, the sort/merge, and the hashed, or bucket, join (14). These approaches either use brute force and compare

all tuples of both relations, or use some method to group the tuples in a manner which reduces the total number of compares.

The nested-loop algorithm is the simplest and least efficient method as it involves the comparison of all tuples of one input relation to all tuples of the other input relation. As the title indicates, it is implemented in a nested-loop control structure that is of order $O(nm)$, where n is the size of one input relation and m is the size of the other. Although the nested-loop is a very inefficient algorithm for equi-join processing, it is the only join algorithm that is insured to work for all possible join criteria (12:117).

The sort/merge algorithm uses one of the grouping methods to decrease the number of compares that are required to perform the join operation. The sort/merge first sorts both of the input relations on the join field. It then uses a merge operation to compare the tuples of the relations. If the input relations are already ordered on a field other than the join field, the relations must still be sorted as unordered relations. The sort/merge is the most efficient sequential method for performing equi-join operations (12:121).

The bucket join algorithm also groups the tuples from the input relations to decrease the number of compares that must be performed. It accomplishes the grouping by performing a common hash function on the join field of the input relations. The result of the hashing process is a number of corresponding buckets from each of the relations that can then be joined to produce the result relation. The same hash function must be applied to each of the tuples in each of the relations to insure that joining of the corresponding buckets will result in the entire result relation. (12:208).

Kearns' models include many variations of the parallel versions of each of these algorithms, as well as other algorithms. For this thesis one variation of the parallel nested-loop join algorithm and one variation of the parallel bucket join algorithm were selected for implementation and analysis.

3.4.1 Parallel Nested-Loop Join Implementation The implementation of any parallel algorithm involves the composition of a set of sequential algorithms on the nodes and a communication structure that allows the nodes to communicate. The parallel nested-loop join algorithm consists of a sequential nested-loop algorithm running on the nodes which

performs the required comparisons on multiple fragments of the input relations concurrently. This approach is referred to as *data parallelism* as the same operation is applied to different sets of data on the various processors. This implementation employs a ring communication structure to allow the relation fragments to be passed among the processors.

3.4.1.1 Node Processing The algorithm applied on each of the nodes is a simple sequential nested-loop join which compares all of the tuples from a fragment of one relation to all of the tuples from a fragment of the other relation. Ullman shows in (18) that the most efficient way to perform the nested-loop join is to first fill the available memory with blocks of the smaller relation. Then read blocks of the larger relation one at a time, joining each with the memory resident blocks from the smaller relation. Each of the blocks of the smaller relation will only be read once, however, if the entire relation will not fit into memory, multiple memory fills will be required. Each of the blocks of the larger relation, on the other hand, will be read every time memory is filled with blocks of the smaller relation.

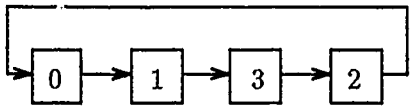
This approach is easily implemented in the parallel environment. Each processor first fills its memory with local blocks of the smaller relation. Local blocks are those resident on a processor node's associated disk node. Each processor then reads one local block from the larger relation and joins it with the blocks resident in memory. When each processor finishes the join operation, it passes the current larger relation block to the next node in an embedded ring structure. Here the block is joined with the memory resident smaller relation blocks and then passed on to the next ring node. This process continues until the larger relation blocks have been processed by all of the nodes in the ring. At this point, each processor refills its memory with local blocks from the smaller relation. This continues until all of the blocks from the smaller relation have been loaded into memory and joined with all of the blocks from the larger relation. Another block of the node memory is also used to collect the result relation tuples as they are generated. When the result buffer fills up, its contents are sent to the host processor to be merged into the final result.

3.4.1.2 Communication Structure Different communication structures can be implemented to allow the processor nodes to exchange blocks of the larger relation. Cloud

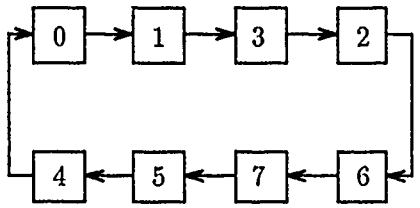
(4) presents a discussion and comparison of the ring structure and the broadcast structure structure for the MARK III hypercube. In his models, Kearns assumed a broadcast communication structure for the nested-loop algorithm. The iPSC/1, however, does not provide an efficient broadcast capability between the nodes in the cube. It is possible to simulate a broadcast through the use of a spanning tree however. When implemented using broadcast communication, the nested-loop join algorithm is slightly different than when the ring communication structure is used. For the broadcast algorithm, each processor node fills its memory with local blocks from the smaller relation in a manner similar to the ring algorithm. To disseminate the blocks of the larger relation, however, the processor nodes must broadcast one at a time. Therefore, a join operation performed on n processor nodes with m larger relation blocks will require $m * n * logn$ communications. The nested-loop utilizing the ring structure, on the other hand, will only require $m*(n-1)$ communications. Therefore the ring structure was selected for this implementation. The ring structure is implemented for different numbers of nodes as shown in Figure 3.4. Each of the neighbors of a particular node in this configuration passes data only to nearest neighbor nodes and hence avoids multiple hop communications.

3.4.2 Parallel Bucket Join Implementation Unlike the parallel nested-loop algorithm which uses the brute force method to join the input relations, the implemented parallel bucket join algorithm first groups the data in order to decrease the actual number of compares that must be performed. The bucket join is actually performed in two separate steps. The first step performs a hash function on each of the input relations and the second step performs the join processing. All of the specified number of nodes are used to perform both the hash step and the join step. Both of these functions are performed using data parallelism.

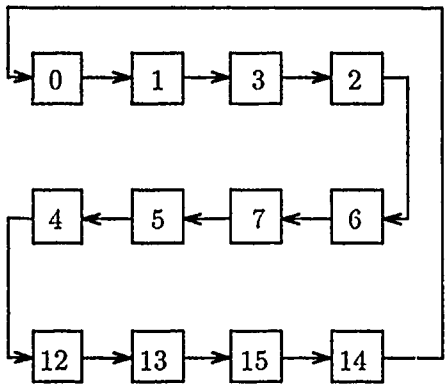
3.4.2.1 Hash Step The purpose of the hash step is to group the tuples from each of the input relations into a number of buckets equal to the number of processors available to perform the join operation. This will allow each processor to perform one sequential join operation on corresponding buckets from each of the relations. To accomplish this grouping, the same hash function must be applied to the join attribute of both input



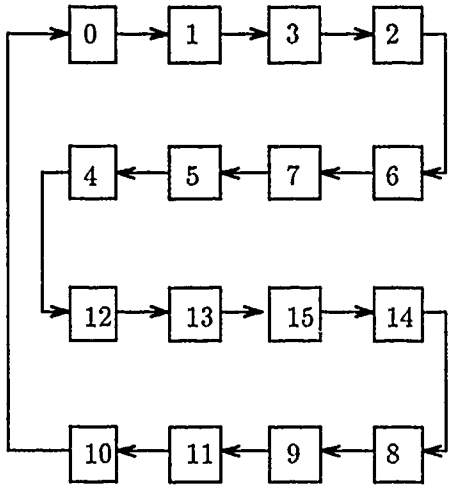
a) 4 Node Ring



b) 8 Node Ring



c) 12 Node Ring



d) 16 Node Ring

Figure 3.4. Nested-Loop Communication Structures

relations. The best performance in the join step will be achieved when the hash function uniformly distributes the tuples across the available buckets. Kearns' models make the assumption that this uniform distribution is always achieved by the hash step. The hash function chosen for this implementation simply divides the value of the join field by the number of processors to be used for the join operation. Assuming that input data is equally distributed across a given range and that the number of processors will divide evenly into that range, this function will result in a uniform distribution.

The hash program that runs on each node sets aside a memory buffer for each of the buckets. It then loads the remainder of memory with blocks of tuples from the relation

being hashed. It then processes the tuples sequentially, performing the hash function on the join attribute of each tuple and then writing it to the proper bucket buffer. When a bucket buffer fills up, it is immediately sent to the proper processor node and stored to the associated disk node. This process logically assumes a fully connected communication structure which allows direct communication between all processors. However, the cube structure will actually require some of the hash messages to pass through intermediate nodes to reach the desired destination.

3.4.2.2 Join Step In the join step of the bucket join algorithm, each processor simply performs a sequential join operation on the intermediate relations that were previously stored on the corresponding disk node during the hash step. Any of the sequential methods can be applied to accomplish this step. Kearns' bucket join models included using the nested loop algorithm and also the sort/merge algorithm during this step. The implemented version of the bucket join contains only the nested loop join. It is implemented in much the same manner as it was for the parallel nested loop algorithm. However, there is no communication between the nodes during the join step.

IV. Single Step Implementation Analysis

The analytical models developed by Kearns provide a means for predicting the performance of parallel relational database queries for a variety of machine architectures and data storage structures. To attempt to validate these models, they must be implemented and tested, with the results being compared with the model projections. The approach chosen to evaluate the nested-loop and bucket join models was to develop a series of test cases to be run on both the models and the actual implementations on the iPSC/1. The following sections first present in detail Kearns' models for the algorithms. They then document the application of standard parallel analysis techniques to both the modeled and the measured runtime data, giving insight into possible inconsistencies between the two. Based upon the results of the runtime analysis, the models are then examined further in an attempt to identify the reasons for the inconsistencies. A brief description of all of the test cases and the associated data is documented in Appendix B.

4.1 Parallel Performance Measures

When analyzing the performance of programs on concurrent computers, there are some basic measures that provide insight into the performance improvement achieved through parallelization of the process. The first of these is speedup (S). Speedup is defined as the ratio of the time required to perform a given function on a single processor to the time required to perform the same function on multiple processors (6:55). Given a time T(N), where N is equal to the number of processors, speedup is expressed as

$$S = \frac{T(1)}{T(N)}$$

Another performance measure that is closely related to speedup is concurrent efficiency (ϵ) (6:55). Concurrent efficiency is a measure of how well the concurrent implementation is utilizing the total processing capability of the parallel architecture. Concurrent efficiency is expressed as

$$\epsilon = \frac{S}{N}$$

where N is equal to the number of processors. A parallel implementation that yields a speedup of N has a concurrent efficiency equal to 1.

Ideally, the speedup realized when a process is spread over N processors should be N. In reality, however, a speedup of N is difficult to achieve unless some hueristic is applied to actually reduce the amount of processing that must be performed. This reduction must be realized in both the serial portion of the code as well as the additional code required to parallelize the process. There are overhead costs that are associated with the parallelization of a process. Overhead (f) can be expressed as

$$f = \frac{1}{\varepsilon} - 1$$

According to Fox (6:55-56) there are four factors that limit the ability to achieve a speedup of N. They are algorithmic overhead, software overhead, load balancing, and communication overhead. Algorithmic overhead results when it is impossible to find a parallel algorithm that can solve a problem as efficiently as the sequential algorithm. Software overhead may result when an equivalent parallel algorithm is found, but it requires additional processing when it is split among multiple processors. Load balancing involves spreading the amount of processing evenly among the concurrent processors. The process can only run as fast as the slowest node. If the processing cannot be spread perfectly across all processors, overhead time results. Communication overhead is the time spent in communication between processors. Any time during the execution of the concurrent version of a program that a processor must stop to communicate with another processor, overhead time results.

4.2 Nested Loop Join Analysis

4.2.1 Nested Loop Model Analysis The nested loop-join algorithm is examined first. The performance parameters for the models are listed in Table 4.1. For this model Kearns assumes that the input relations, R and S, are both spread uniformly across the available disks and that each disk can broadcast a block of data to all processor nodes in a constant time T_{io}. This broadcast capability provides the means for each node to send each block of the smaller input relation, S, to all other nodes. Each processor is then able to join its

Table 4.1. Model Performance Parameters

T_c	query compile time (0 ms)
T_m	time to send a message between backend and host (0 ms)
T_d	average disk access time (variable)
p_b	memory blocks per processor (variable)
p	number of processors (variable)
d	number of disks (variable)
b	blocks per track on disk (10)
T_s	seek time of one track on disk (10 ms)
T_{io}	transfer time from disk for one block of data (variable)
R	number of blocks in R relation (variable)
S	number of blocks in S relation (variable)
J_B	number of blocks in join result relation (variable)
T_{bt}	time to send one block of data between backend and a node (16 ms)
T_{sc}	time to scan one block of data (7.5 ms)
T_b	time to process a block with a join operation (145 ms)
jsf	join selectivity factor (variable)

local fragment of the R relation with the entire S relation. This second assumption is not consistent with the implementation on the iPSC/1. The iPSC/1 implementation organizes the processor nodes into a logical ring structure that is used to pass the blocks of the larger relation to all of the processor nodes. This inconsistency provides the opportunity to examine the accuracy of the general model for a case where the hardware implementation does not exactly match the assumptions made for the model. Based upon Kearns' assumptions of the uniform distribution of R and S across the available disks and the capability to broadcast the S relation from the disks to the processing nodes, the performance model for the nested-loop join algorithm is (12:160):

$$T_c + T_m + [T_d + (((p_b * p)/d)/b) * T_s + (((p_b * p)/d) * T_{io})] * (R/(p_b * p)) + ((R/p) * S * T_b) + [(S * (R/(p_b * p))) * T_{io}] + [(j_B/d) * T_{bt}] \quad (4.1)$$

Based upon the assumption that the number of disks is always equal to the number of processors, this equation can be simplified to:

$$T_c + T_m + [T_d + (p_b/b) * T_s + p_b * T_{io}] * (R/(p_b * p))$$
$$+((R/p) * S * T_b) + [(S * (R/(p_b * p))) * T_{io}] + [(j_B/d) * T_{bt}] \quad (4.2)$$

A more indepth discussion of the model parameters and the single step models is provided in Appendix A.

4.2.2 Nested-Loop Runtime Analysis The first step in the comparison of Kearns' nested-loop join model to the implementation on the iPSC/1 is to consider the overall runtime of the two. Tests were run for three different problem sizes in order to expose any sensitivities that may exist with respect to problem size. For these initial runtime measurements, all other performance parameters were held constant while the input relation sizes were set at 48, 96, and 144 blocks. Figures 4.1, 4.2, and 4.3 show graphically the theoretical runtimes versus the observed runtimes. Each of the figures illustrates that the difference between the theoretical and observed runtimes is smaller for smaller numbers of nodes than for larger numbers of nodes. Comparing the figures shows that the difference between the theoretical and observed runtimes is also smaller for larger problem sizes than for smaller problem sizes. The lines representing the theoretical and observed runtimes get closer together as the problem size increases from Figure 4.1 to Figure 4.2 to Figure 4.3. The average difference between the theoretical and observed runtimes for the 48 block input relation case was about 8.5 %. The average difference for the 144 block input relation case was 1 %.

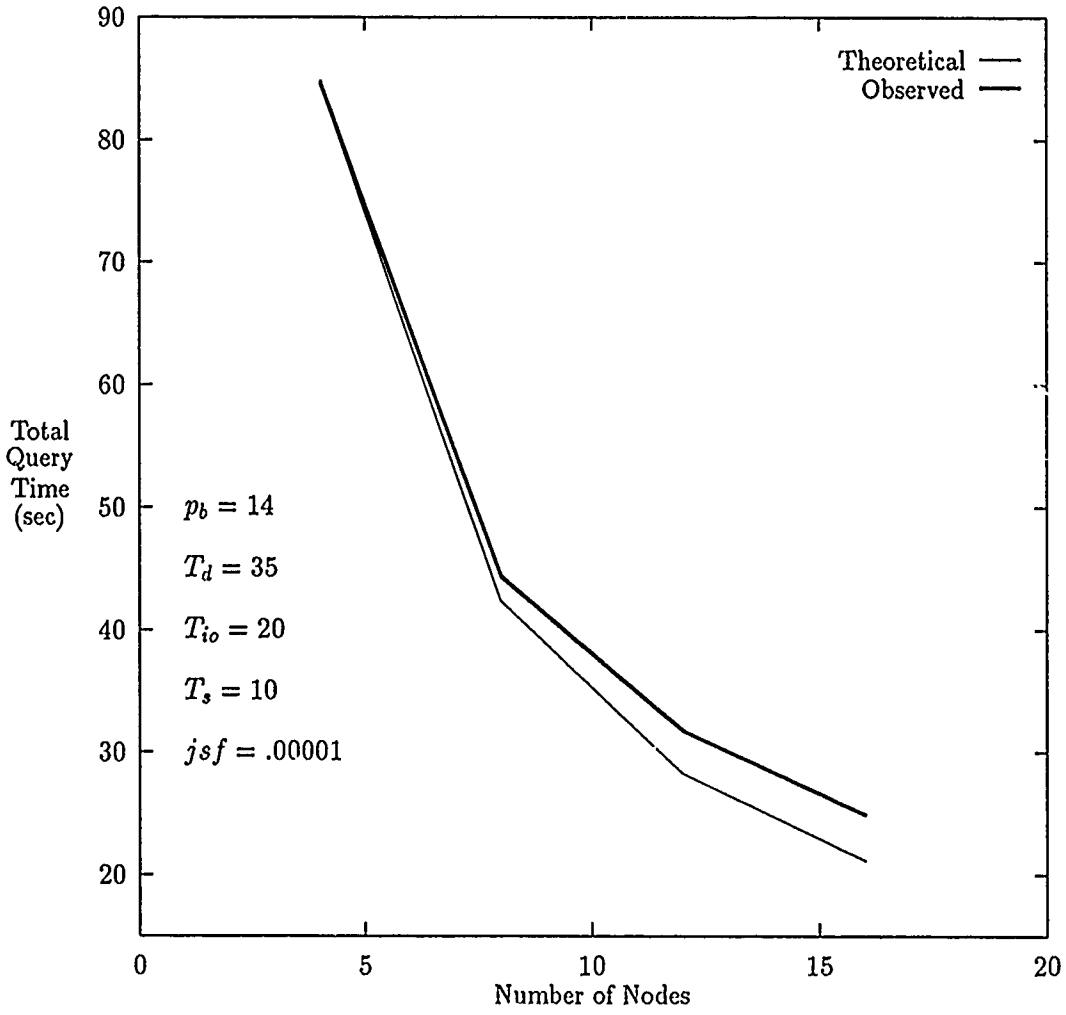


Figure 4.1. Nested-Loop 48 Blocks x 48 Blocks

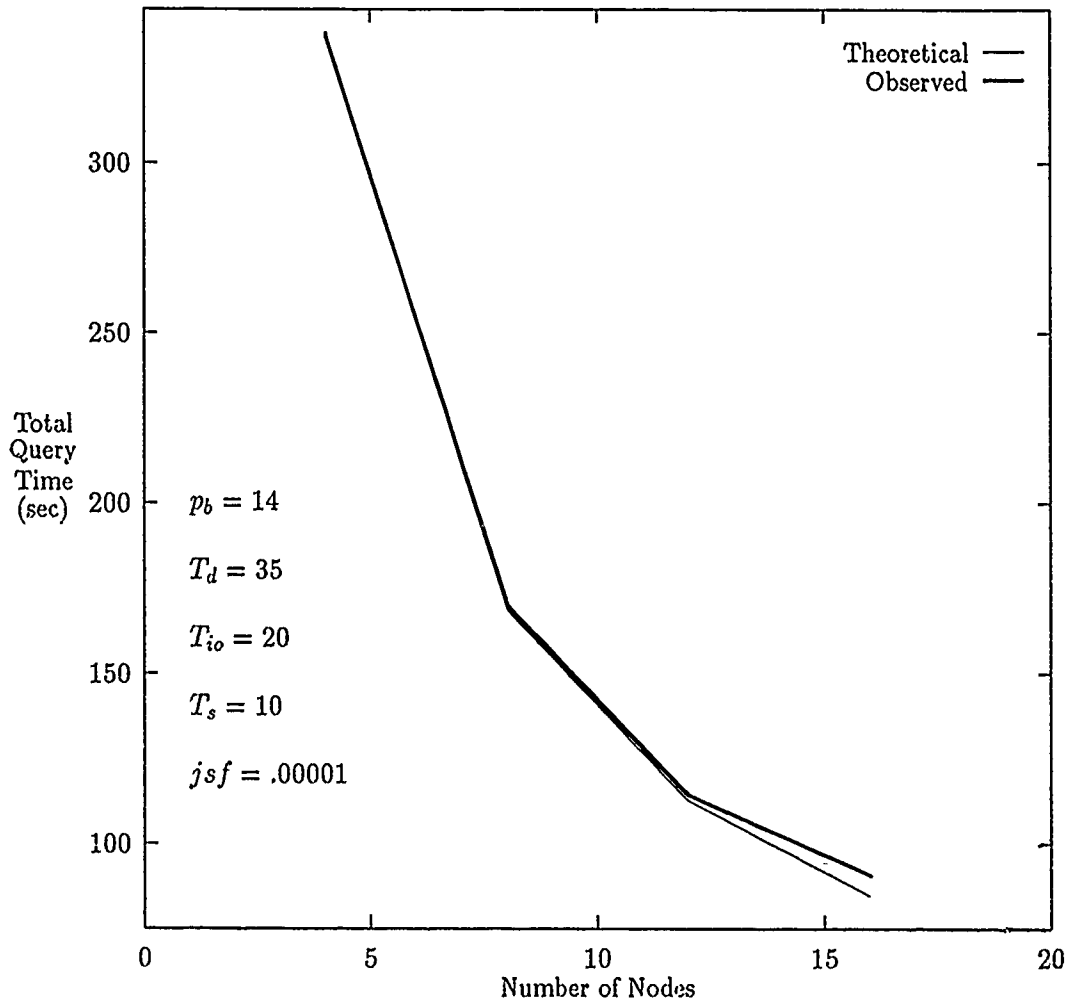


Figure 4.2. Nested-Loop 96 Blocks x 96 Blocks

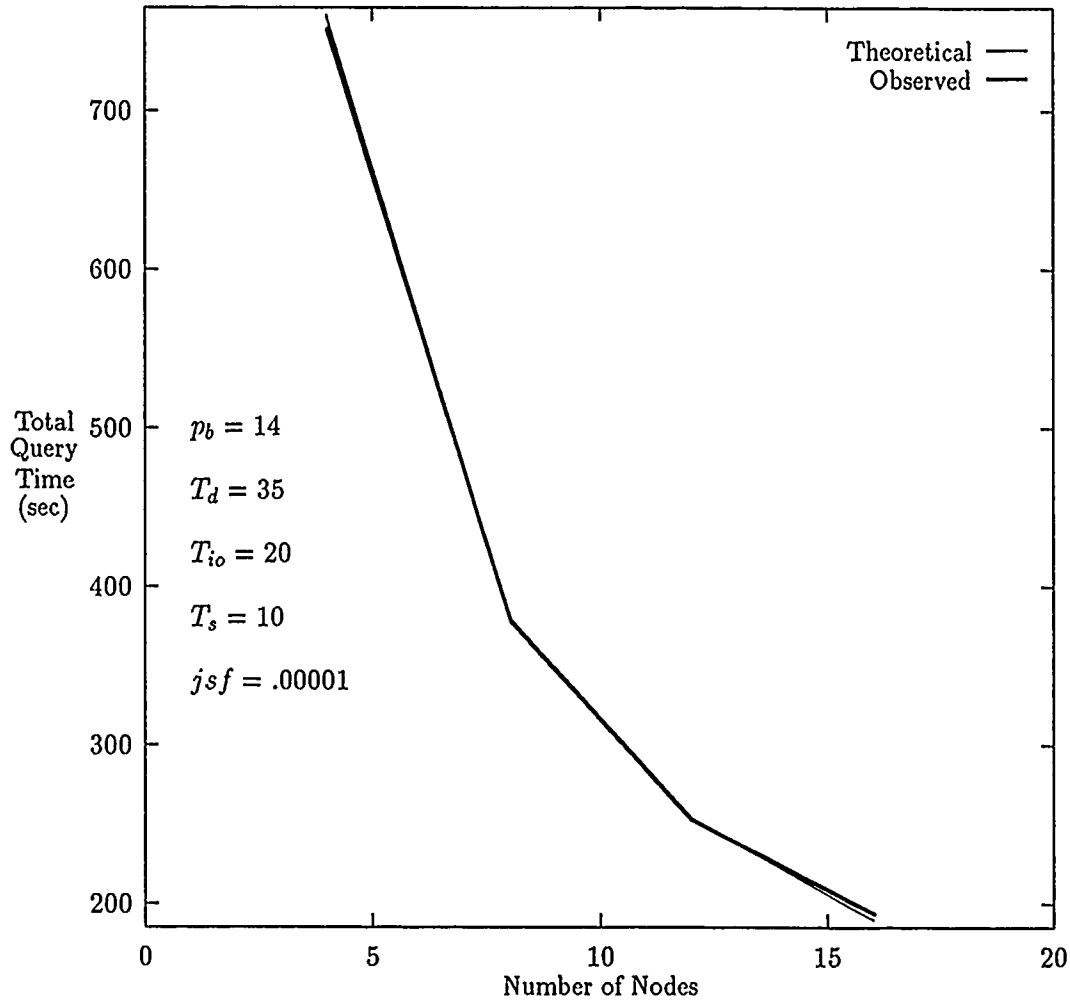


Figure 4.3. Nested-Loop 144 Blocks x 144 Blocks

Further insight can be gained through examination of the speedup (S) achieved as the number of processor nodes is increased for each problem size. Figures 4.5, 4.6, and 4.7 graphically show S for each problem size. The speedup graphs show that the model predicts virtually linear speedup for each of the three problem sizes. The observed runtimes, on the other hand, reflect close to linear speedup up to a point at which time the measured curve begins to level off. It has been documented that when the number of concurrent processors applied to solve a fixed size problem is continuously increased, a point will be reached when the amount of overhead begins to dominate speedup (6:59). When this occurs, the speedup realized with each additional node actually decreases. This point is often specified in terms of the ratio of communication time to calculation time (t_{comm}/t_{calc}). Figure 4.4 illustrates this point. The graph shows that as the number of processors, N, is increased, t_{calc} decreases. At the same time, t_{comm} is increasing at some rate . As the two curves begin to converge, the speedup curve will begin to flatten and then the speedup will eventually begin to decrease.

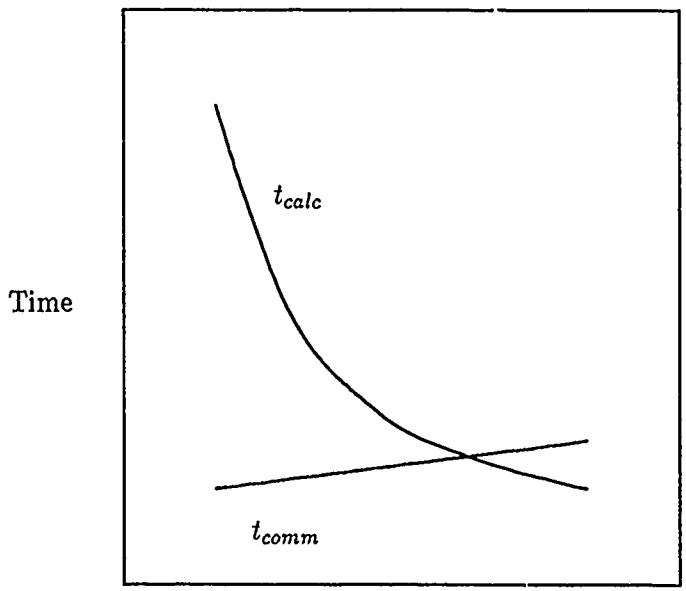


Figure 4.4. Graph of t_{comm} and t_{calc}

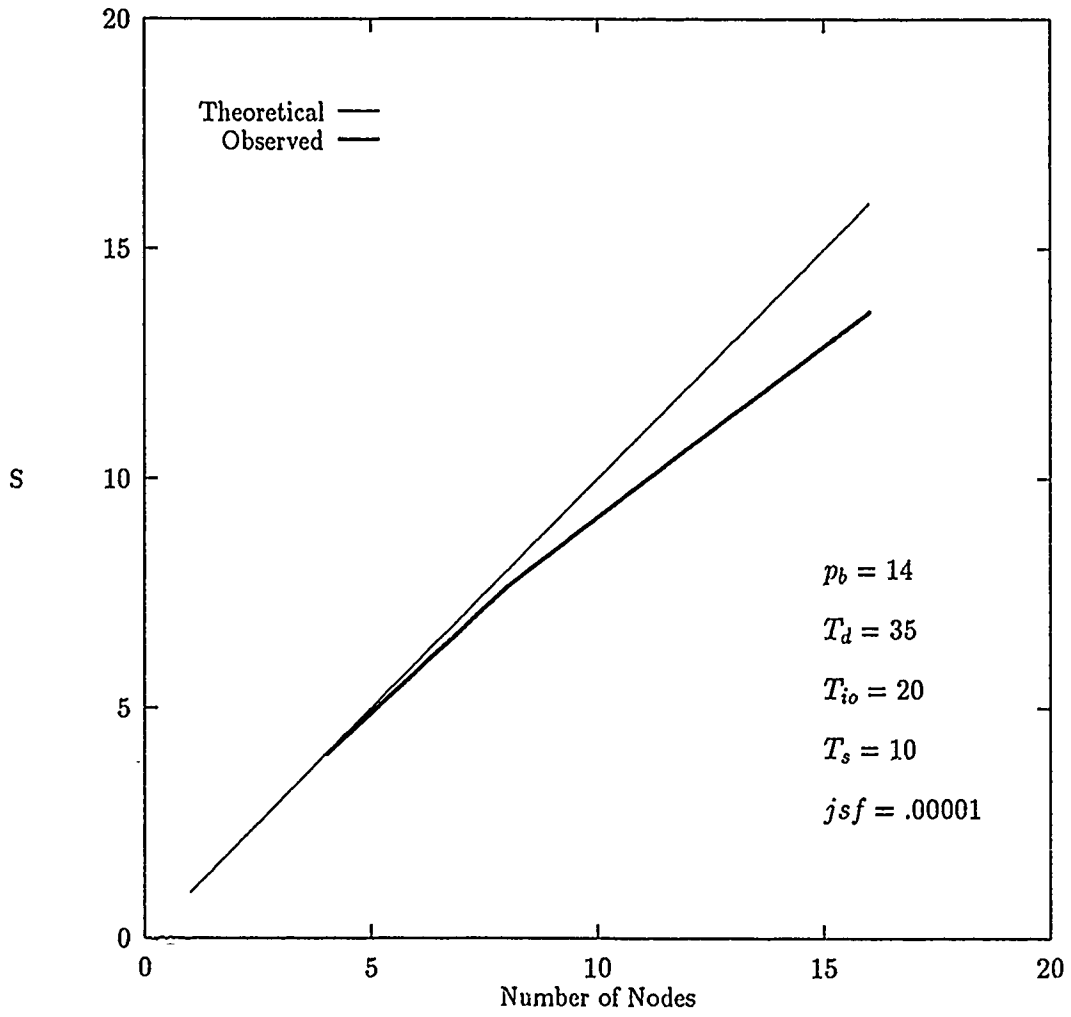


Figure 4.5. Nested-Loop Speedup 48 Blocks x 48 Blocks

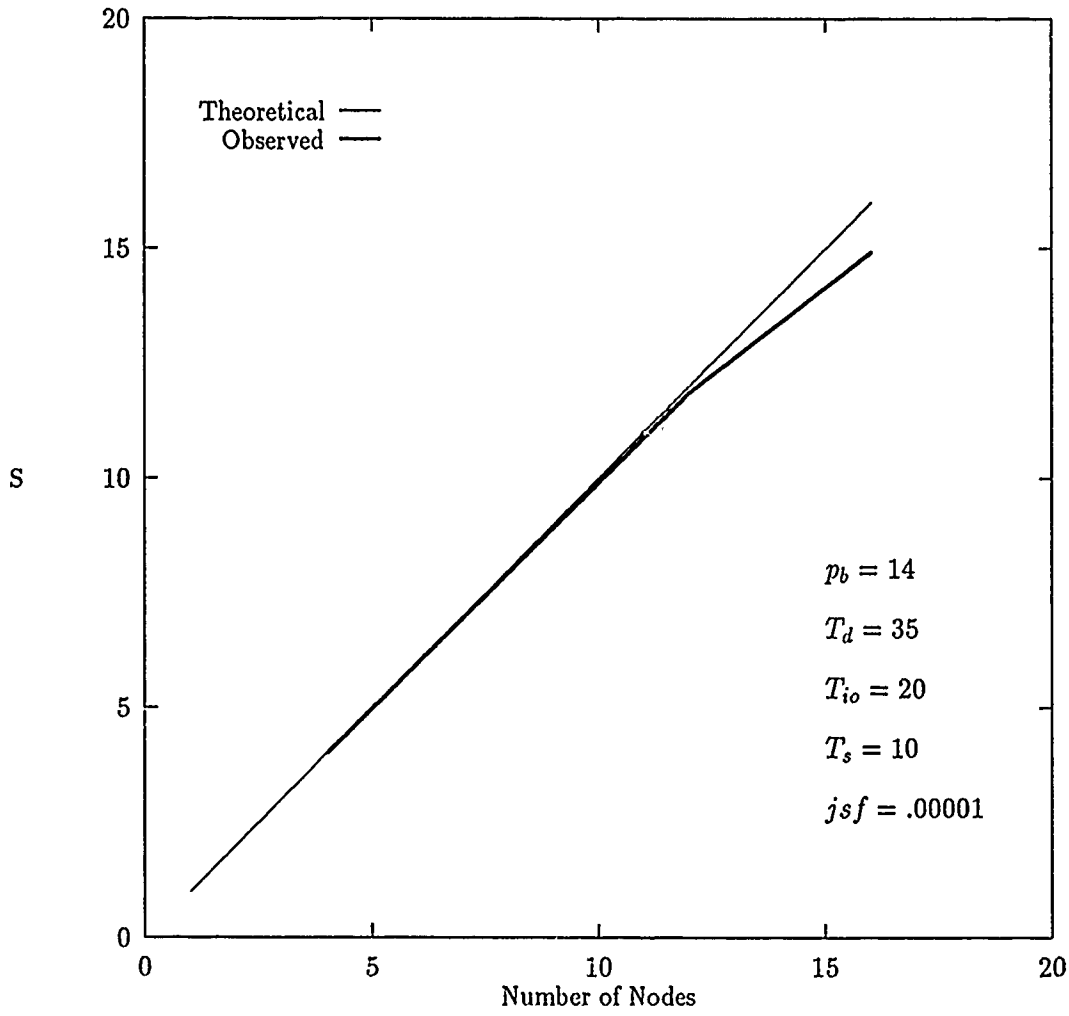


Figure 4.6. Nested-Loop Speedup 96 Blocks x 96 Blocks

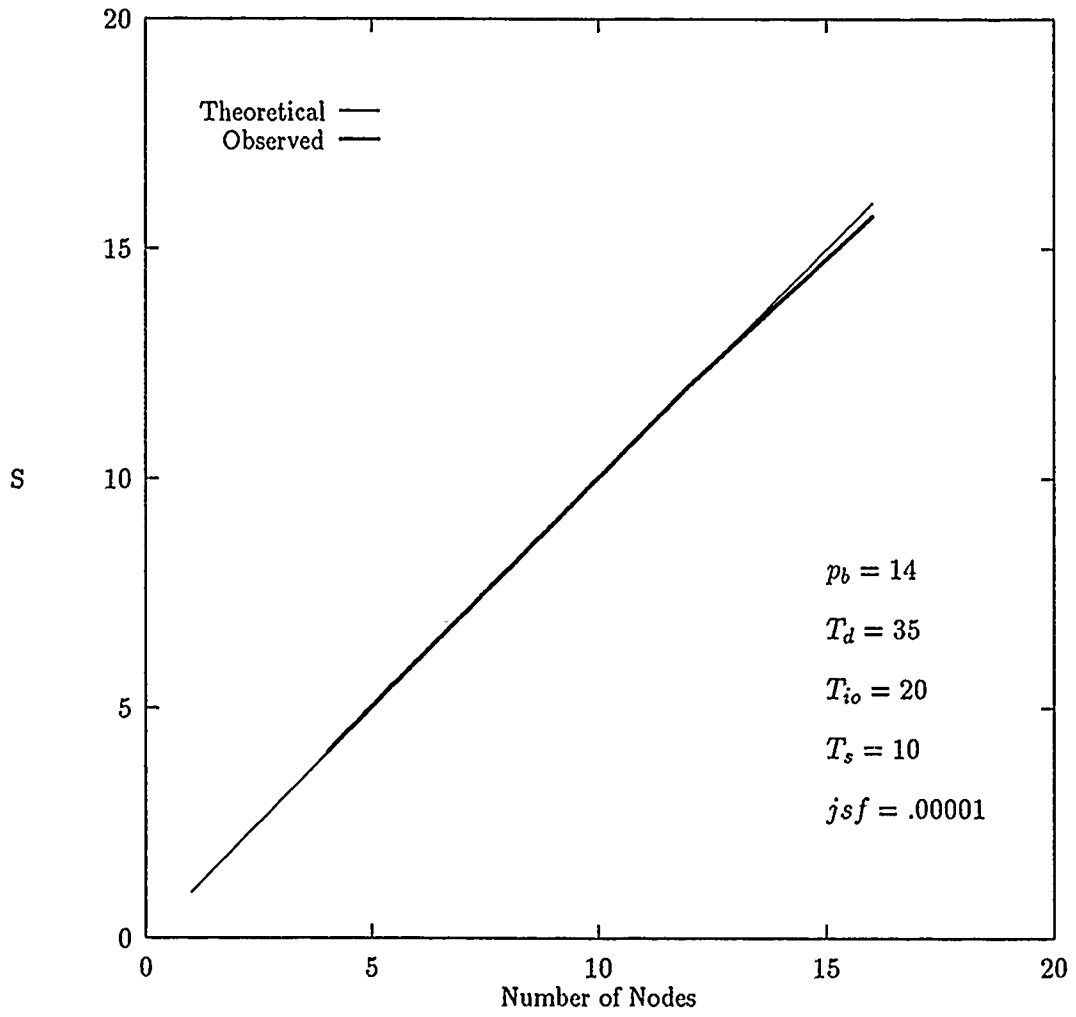


Figure 4.7. Nested-Loop Speedup 144 Blocks x 144 Blocks

Further investigation of the nested-loop model reveals that it does not actually predict linear speedup for all values of N. There is a point at which the modeled speedup curve begins to flatten. The value of N at which this occurs, however, is significantly greater than 16. This is illustrated in Figure 4.8. This graph shows the modeled speedup with the

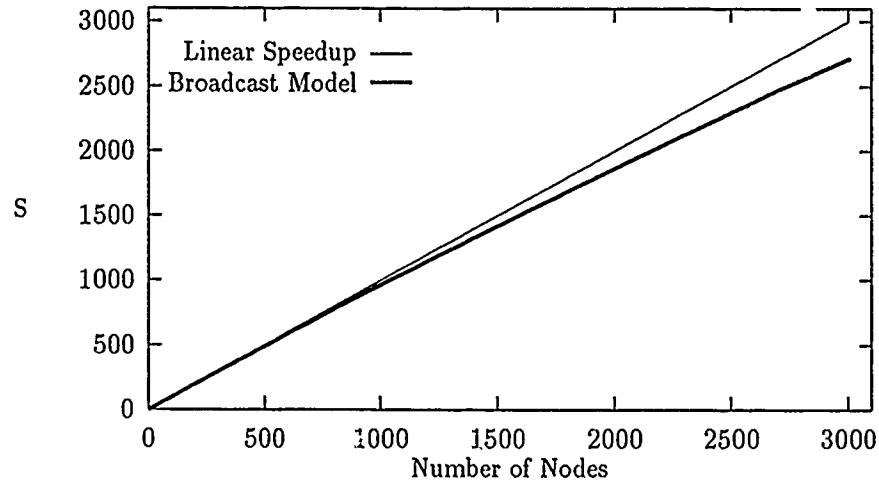


Figure 4.8. Theoretical Speed for Very Large N

value of N increasing to 3000 nodes. Even with N equal to 3000, the speedup curve has not flattened significantly. However, it does show that the curve is beginning to flatten.

The difference in the speedup curves of Kearns' model and the iPSC/1 implementation suggests that as N is increased, the communication overhead is actually growing at a faster rate than the model predicts. As stated earlier, Kearns assumed that the blocks of the smaller relation could be disseminated to the processor nodes through a series of broadcasts from the disks. The broadcast could not be done efficiently on the iPSC/1, however, and so the ring communication structure was implemented instead. The time to perform the ring communication during the nested-loop algorithm can be expressed analytically as

$$[(p-1) * \lceil R/(p_b * p) \rceil * (S/p) * T_{io}] + [(\lceil R/(p_t * p) \rceil * (S/p)) * (T_d + T_{io})] \qquad (4.3)$$

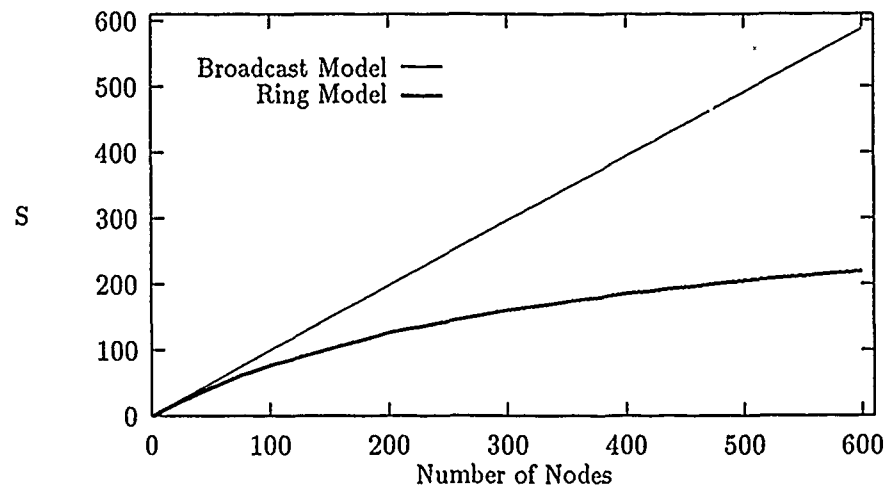


Figure 4.9. Broadcast Model Speedup Versus Ring Model Speedup

Figure 4.9 illustrates the speedup curve of the nested-loop model with the ring communication expression as opposed to Kearns' original model containing the broadcast communication expression. The speedup curve of the ring model flattens out with N well under 100. However, it does not flatten quite as soon as the measured speedup from the iPSC/1 implementation, as shown in Figure 4.10. This is because there are additional nondeterministic overheads that are not accounted for in the model. The average difference between the theoretical and observed runtimes using the ring communication model is about 6 % for the 48 block input relation case and about 1 % for the 144 block input relation case.

4.3 Bucket Join Analysis

4.3.1 Bucket Join Model Analysis The bucket join implementation on the iPSC/1 is very similar to the algorithm modeled by Kearns. Kearns assumes that the input relations are both uniformly spread across the available disks. The bucket join is then performed in two distinct steps. The first step is the hash step. During the hash step, each node applies a common hashing function to all tuples of each local relation fragment. Based upon the the results of the hash function, the tuples are grouped into buckets and each bucket is sent to the proper node. As the bucket blocks are recieved by the nodes they are saved

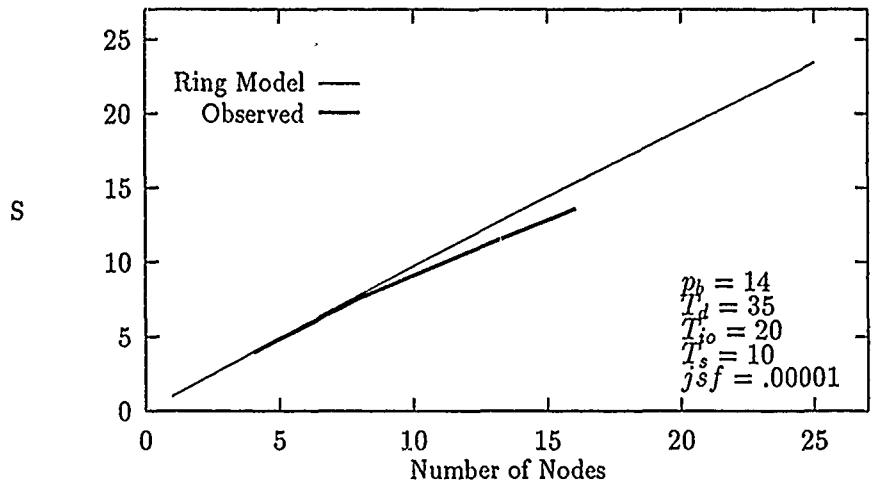


Figure 4.10. Ring Model Speedup Versus Observed Speedup (48 blocks x 48 blocks)

to disk. When the hash step is completed, each node performs a local nested-loop join on the buckets that it has written to disk. The results of the join operation are then sent to the backend processor where they are merged into the complete result relation. The performance model for the bucket join algorithm with the nested-loop is (12:170):

$$
\begin{aligned}
T_c + T_m + T_d + T_{io} + max & \left| \begin{array}{c} (T_{sc} * (R/p)) \\ +2 * [((R/p)+1) * (p-1)] * Tbt \\ or \\ 2 * ((R/d) * Tio) + (((R/d)/b) * T_s) \\ +((R/p)+1) * (T_d + T_{io}) \end{array} \right| \\
+T_d + Tio + max & \left| \begin{array}{c} (T_{sc} * (S/p)) \\ +(2 * [((S/p)+1) * (p-1)] * Tbt) \\ or \\ 2 * ((S/d) * Tio) + (((S/d)/b) * T_s) \\ +((S/p)+1) * (T_d + T_{io}) \end{array} \right| \\
+[(T_d + (pb * T_{io})) * (((R/p)+1)/p_b)] + & (((R/p)+1) * ((S/p)+1) * T_b) \\
& (((S/p)+1) * (((R/p)+1)/p_b)) * T_{io}] + j_B * T_{bt} \qquad (4.4)
\end{aligned}
$$

4.3.2 Bucket Join Runtime Analysis As with the nested-loop runtime analysis, the first step in the comparison of Kearns' bucket join model with the iPSC/1 bucket join implementation is to consider the overall runtimes. With all other performance parameters held constant, tests were run for three different problem sizes with the number of processors varying from 4 to 16. The input relation sizes for the tests were 144, 192, and 240 blocks. The theoretical versus observed runtimes are represented graphically in Figures 4.11, 4.12, and 4.13. The graphs show that the measured runtimes and the runtimes predicted by the model are very close for all cases. The average difference between the theoretical and the observed runtimes for the 144 block input relation case is about 3 %, and the average difference for the 240 block input relation is about 2 %. These results are consistent with the fact that the methods used to implement the bucket join on the iPSC/1 match closely with those modeled by Kearns.

The speedup curves for the bucket join implementation versus the model for the different problem sizes are shown in Figures 4.14, 4.15, and 4.16. The graphs show that increasing the number of processors for this algorithm results in greater than linear speedup, where increasing the number of processors for the nested-loop algorithm resulted in less than linear speedup. This is because the bucket join uses a grouping method to reduce the actual number of compares that must be performed. As the number of processors increases, the size of the buckets decreases and the number of compares decreases. The nested loop, on the other hand, always compares all of the tuples from one input relation with all of the tuples from the other input relation. As expected, the modeled speedup curves and the measured speedup curves match much more closely for the bucket join algorithm than those of the nested-loop algorithm.

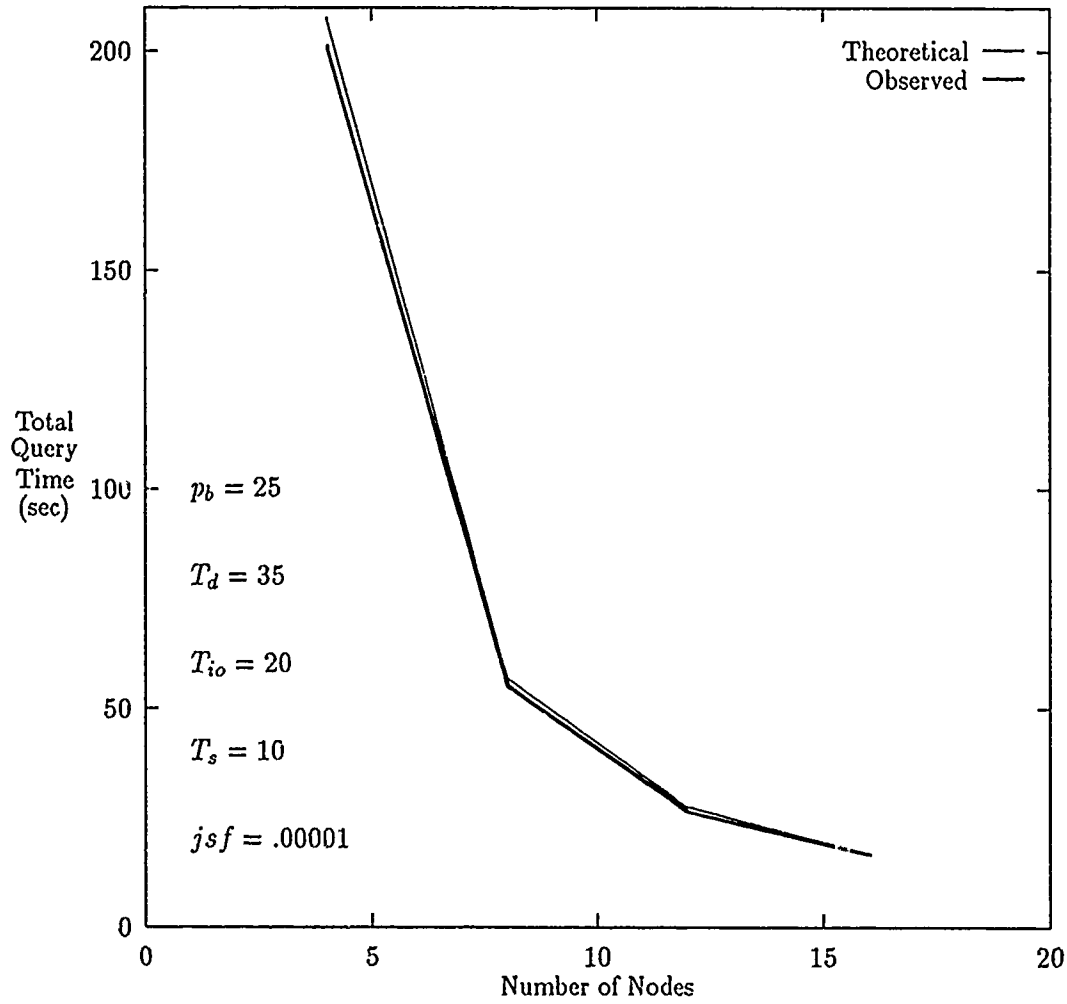


Figure 4.11. Bucket Join 144 Blocks x 144 Blocks

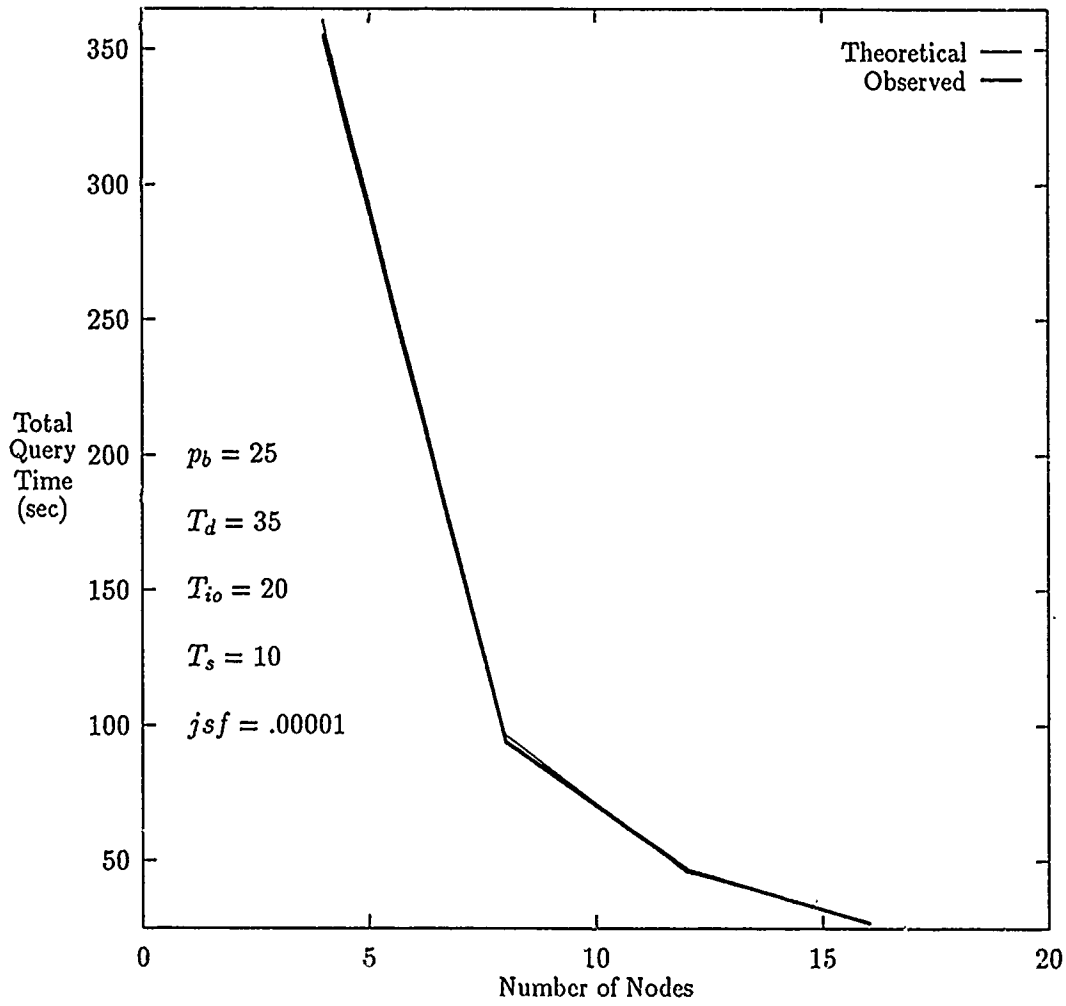


Figure 4.12. Bucket Join 192 Blocks x 192 Blocks

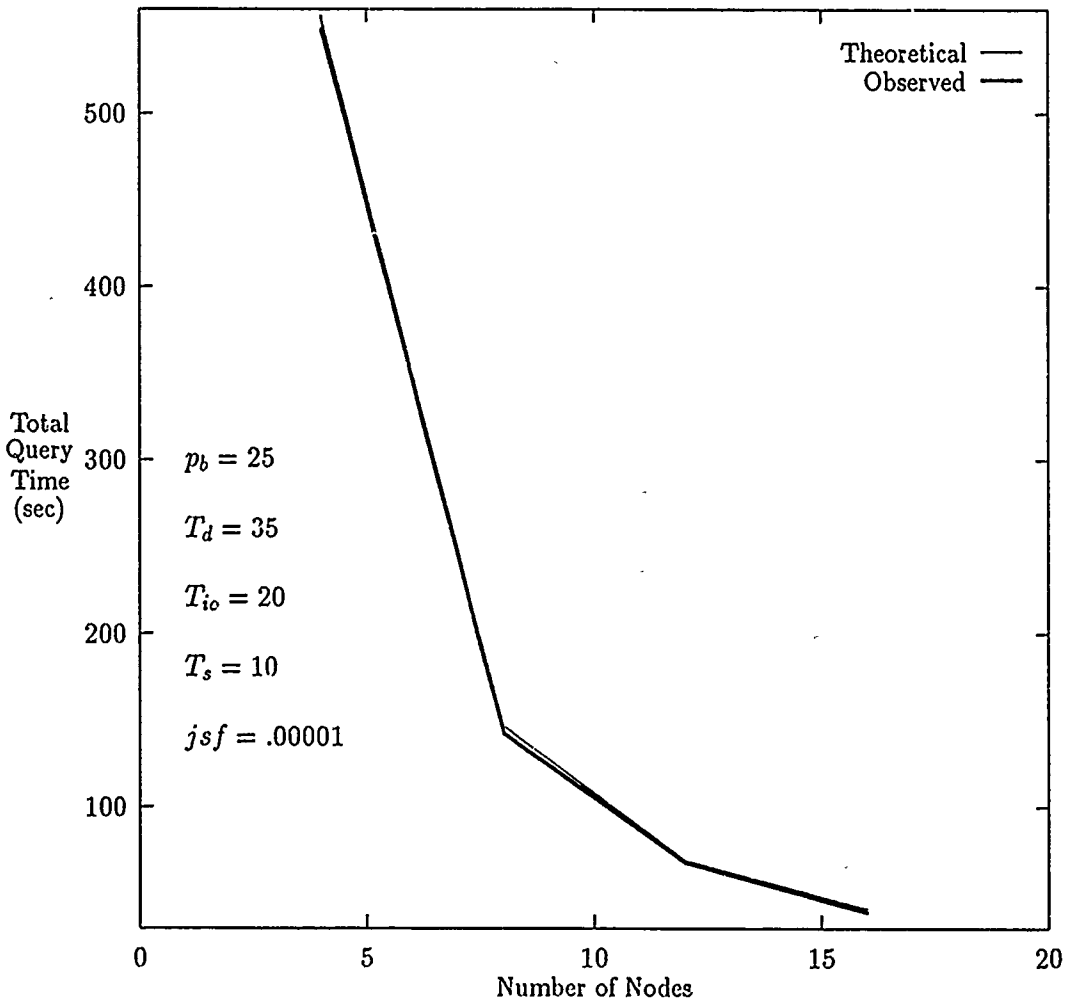


Figure 4.13. Bucket Join 240 Blocks x 240 Blocks

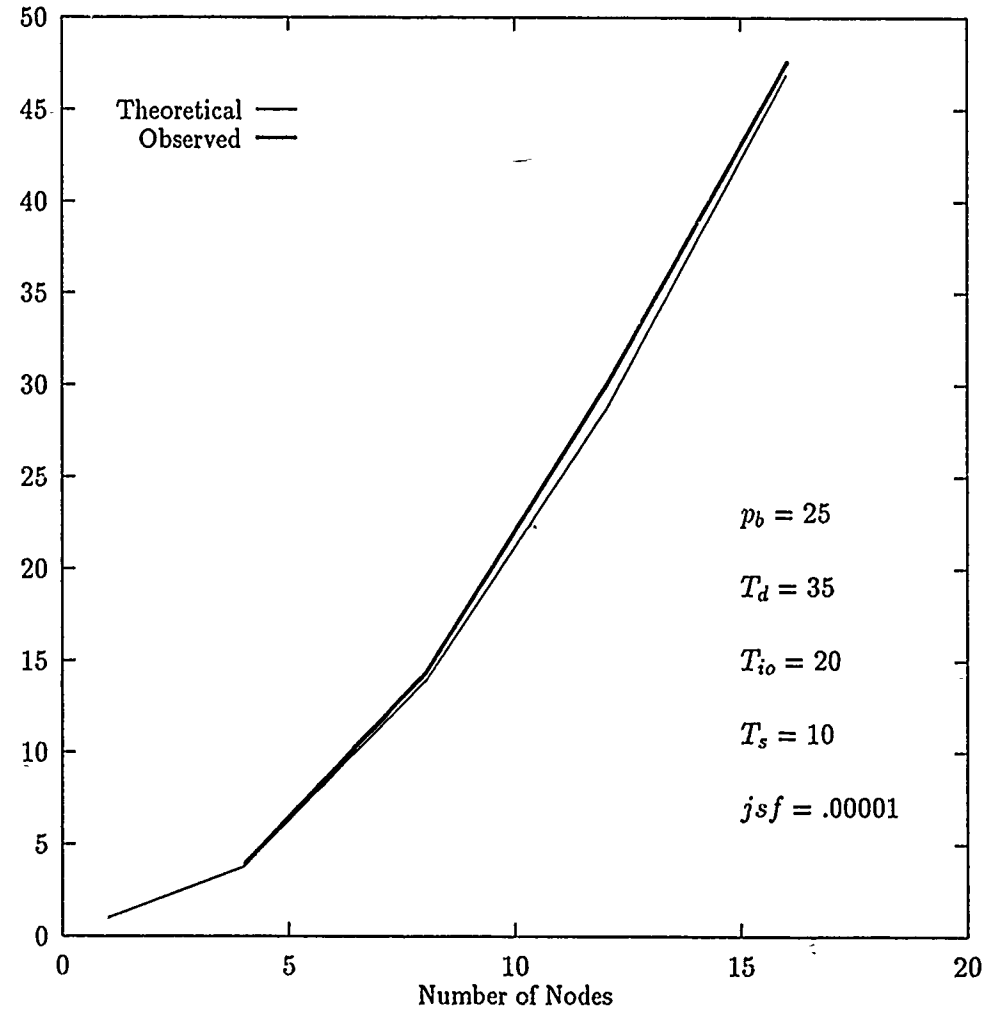


Figure 4.14. Bucket Join Speedup 144 Blocks x 144 Blocks

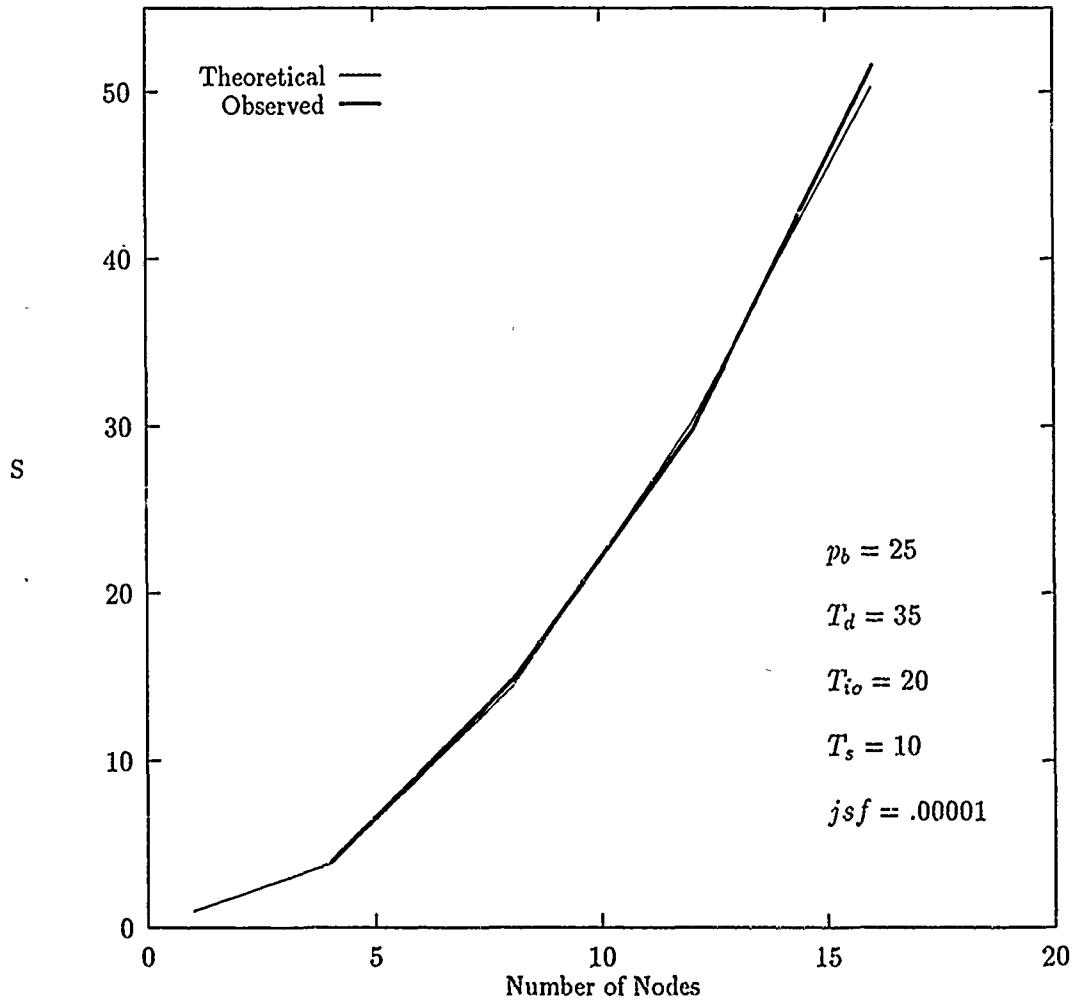


Figure 4.15. Bucket Join Speedup 192 Blocks x 192 Blocks

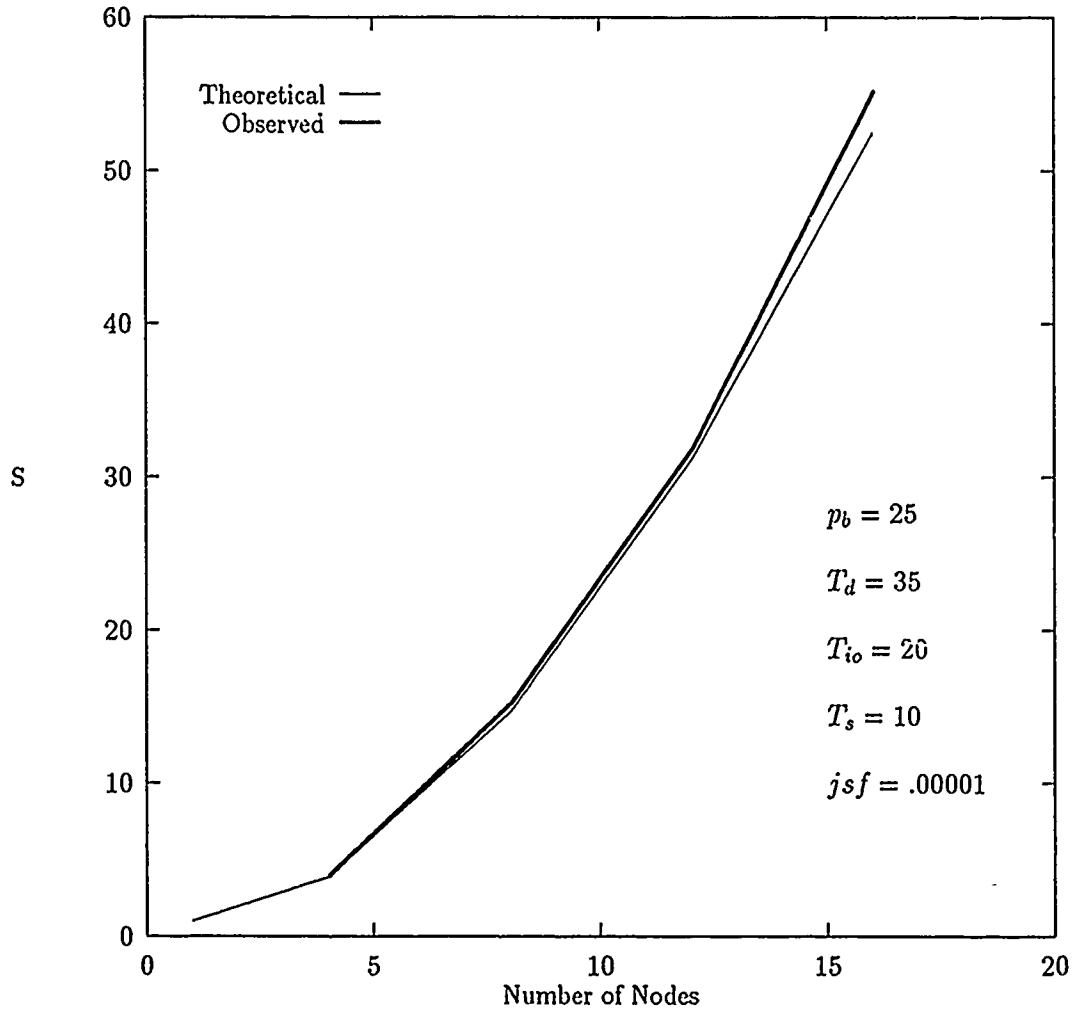


Figure 4.16. Bucket Join Speedup 240 Blocks x 240 Blocks

4.3.3 Bucket Join Sensitivity Analysis The analysis to this point has concentrated on the comparison of the total query time predicted by the models versus the measured total query time over a range of problem sizes and node configurations. An area that has not been addressed is the ability of the models to predict the impact of varying system performance parameters. Tests were first run to evaluate the impact of changing the disk performance parameters. The next set of tests were run to evaluate the impact of changing the join selectivity factor. Both sets of tests were run for relatively small problem sizes and then for relatively large problem sizes with the number of nodes fixed at eight.

For the base performance case, the disk access time, T_d, was set at 35, the block transfer time to disk, T_{io} was set at 20, and the join selectivity factor was set at .00001. These disk performance parameters are hypothetical and do not reflect any particular disk system. The first parameters to be varied were T_d and T_{io}. T_d was set to 70 and T_{io} was set to 40. This simulates a change in the actual disk device and also the communication between the disk and the processor. The result in an increase in average disk I/O response time. Figure 4.17 illustrates the theoretical runtimes and Figure 4.17 illustrates the observed runtimes for the two disk performance levels when measured for relatively small input relations. The theoretical data predicts a virtually constant increase in total query time of about 9 % over this range of problem sizes. The observed data reflects this constant change, although the observed amount of increase in the query runtime is about 13 %. This is slightly greater increase than the model predicted. The analysis of the results of the tests run with relatively large relation sizes yield similar results.

Next the jsf was changed from .00001 to .001. This significantly increased the number of result tuples produced and also increased the time required to process the result blocks. The join of two relations with 104 blocks each, and jsf equal to .00001 will produce about 1180 results. When the jsf is changed to .001 for the same join operation, about 108160 results will be produced. Figure 4.19 illustrates the theoretical runtimes and Figure 4.20 illustrates the observed runtimes for the small problem set sizes. The theoretical data shows that the total query time is increasing at a slightly greater than constant rate as the the size of relation 2 is increased. Examination of the observed data reveals that the trend in the difference between the curves is the same, however, the curve

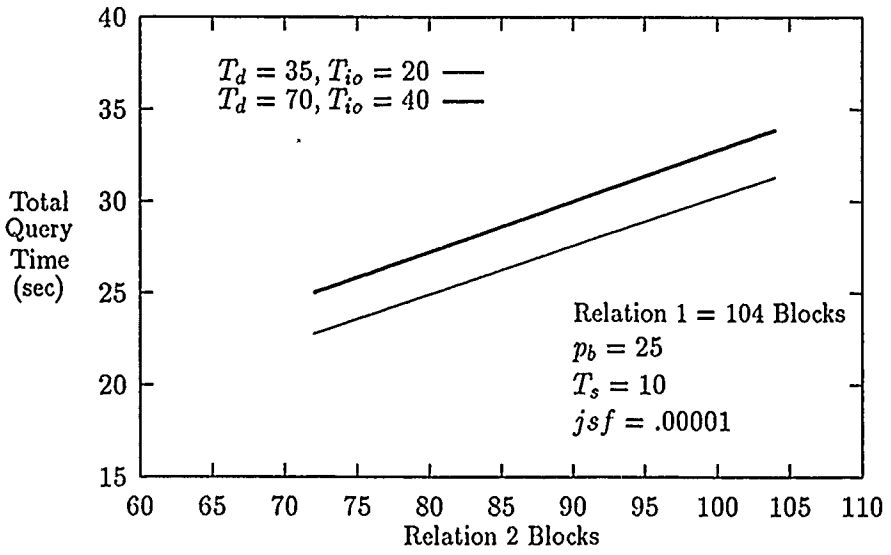


Figure 4.17. Theoretical Performance Impact of Increased Disk Response Time

produced with the jsf equal to .001 has a number of spikes. This is because the data generator produces data which approximates the desired jsf. For some relation sizes it is slightly greater than the desired value and for some sizes it is less. See Appendix E for more information on the actual jsf produced. The spikes do not occur for the .0001 curve because the result relation processing is insignificant due to the small result relation sizes produced. For the test runs in which the actual jsf produced closely approximates the desired jsf, the model predicts about a 13 % average increase in runtime over the range of problem sizes. For these cases, the observed average increase in runtime was about 15 %. The results for the larger relation sizes were similar to these.

4.4 Summary

The single step query models that were analyzed were the nested-loop join and the bucket join using the nested-loop algorithm for the join step. The nested-loop join analysis showed that the runtimes predicted by the model were closer to the observed runtimes for smaller numbers of nodes than for larger numbers of nodes and for larger problem sizes than smaller problem sizes. It was also demonstrated that in cases when the model does

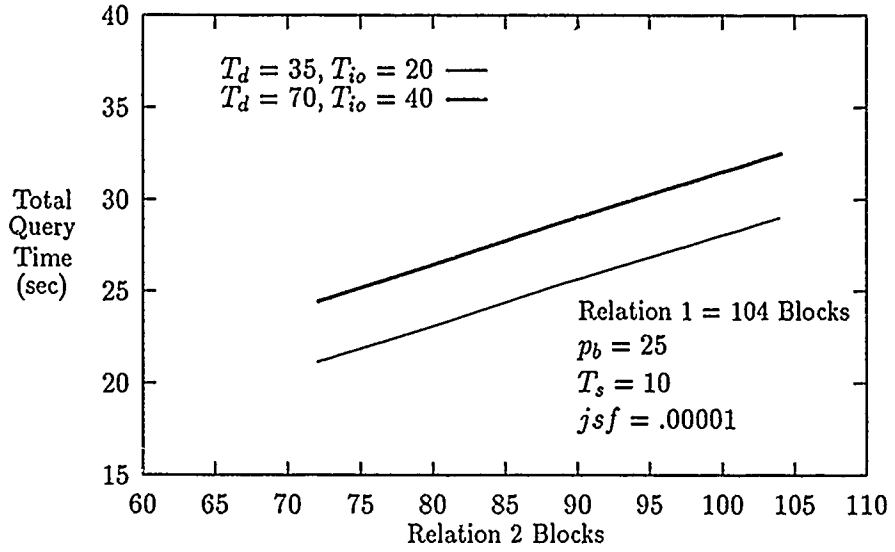


Figure 4.18. Observed Performance Impact of Increased Disk Response Time

not exactly match the implementation, the model can be changed to more accurately reflect the performance of the implemenation. For all test cases, Kearns' nested-loop model with broadcast communication was at least accurate to within about 8.5 %, and the derived nested-loop model with ring communication was at least accurate to within about 6 %. Both models were even more accurate when the input relation sizes were increased.

The results of the bucket join analysis show that the runtimes predicted by the bucket join model matched the observed runtimes closely. For all cases, the bucket join model was at least accurate to within about 3 %. The bucket join sensitivity analysis showed that the model did a pretty good job of predicting the sensitivity of the iPSC/1 implementation to changes in disk response time and jsf. For the disk response time sensitivity test cases, the model predicted an average runtime increase of about 9 % when the disk response time was increased. The observed average runtime increase was about 13 %. For the jsf sensitivity test cases, the model predicted an average runtime increase of about 13 % when the jsf was changed from .00001 to .001. The observed average runtime increase was about 15 %.

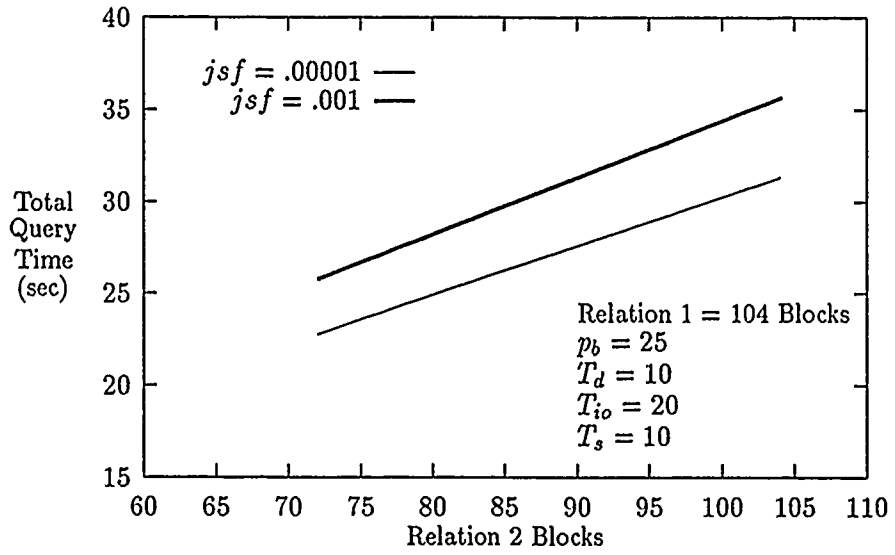


Figure 4.19. Theoretical Performance Impact of Increased Join Selectivity Factor

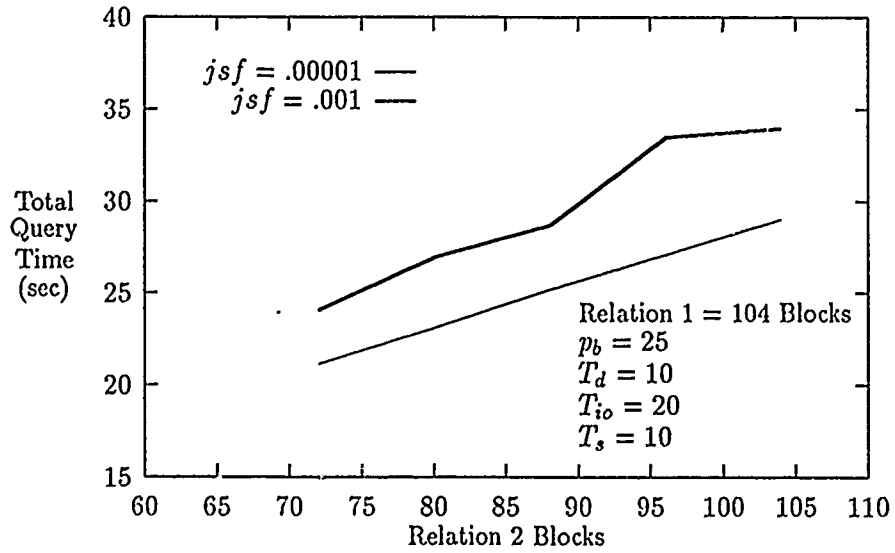


Figure 4.20. Observed Performance Impact of Increased Join Selectivity Factor

V. *Multiple Step Query Model Design, Implementation, and Analysis*

The previous two chapters have dealt with Kearns' single step models in which only one relational database operation is required to perform the entire query. In reality, most database queries require that more than one operation be executed to complete a query. Kearns' multiple step models address these complex queries (12:223). For example, consider the following query:

$$\sigma R \bowtie \sigma S \tag{5.1}$$

To satisfy this query, select operations must first be performed on the R and S relations. The results of these operations must then be joined to produce the desired result relation.

The logical architecture assumed by Kearns in his multiple step models is shown in Figure 5.1. This architecture is very similar to that proposed by Kearns and presented in Section 2.4. It consists of the retrieval layer, which retrieves the input relations from disk and performs any required select-project operations, and the processing layer which performs the more complex binary operations. For the query in equation 5.1, the select operations on R and S would be performed by the retrieval layer nodes and the join operation would be performed by the processing layer nodes. The architecture also contains a backend controller that receives the query from the host processor and initializes the required operations on the retrieval layer and processing layer nodes. The results are then consolidated by the backend controller to be sent back to the host processor if required. The results may also be stored to disk by the retrieval layer nodes.

5.1 Physical Mapping

Like the single step model architecture, this architecture also maps nicely to the iPSC/1 Hypercube. The iPSC/1 cube manager serves as both the backend controller and the output processor. The nodes in the cube are then divided into the retrieval layer and the processing layer. Since each of the retrieval layer nodes requires access to a unique disk,a number of the nodes, equal to the number of retrieval layer nodes, are designated

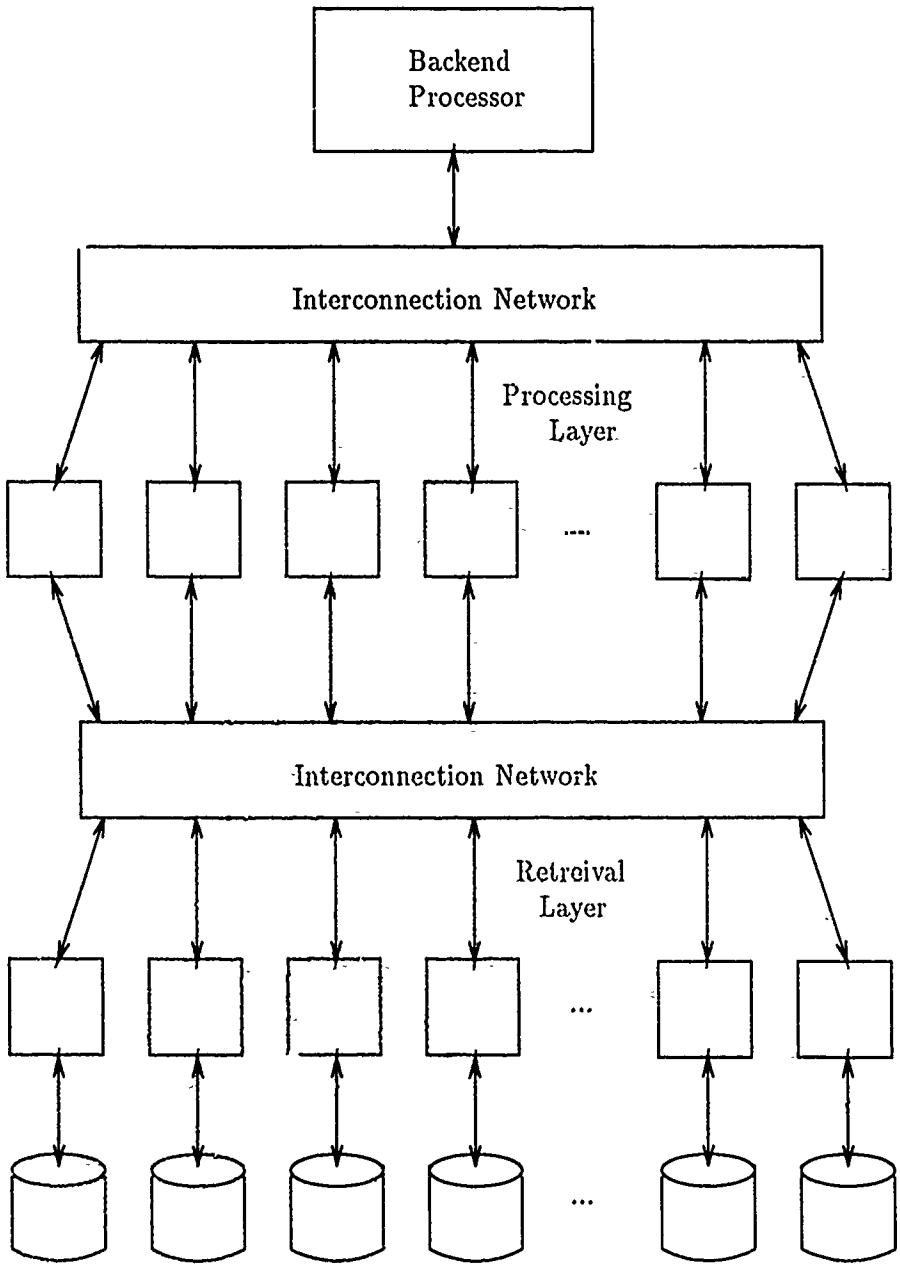


Figure 5.1. Multiple Step Query Architecture (12:282)

as disk nodes. They run the disk simulator program developed for the single step models. The model architecture is also assumed to have secondary storage capability directly connected to the processing layer. However, all of the test cases run on this implementation assume that the entire relations passed from the retrieval layer to the processing layer fit into the memory available on the processing layer nodes. Therefore, disk nodes are not required at the processing layer.

5.2 Multiple Step Bucket Join Implementation

There are two different ways that multiple step queries can be implemented on a concurrent computer. The first is to implement each step utilizing all of the processors that are available in a fashion similar to the single step models (12:239). For this approach, the intermediate relations would have to be stored to disk to be read in by the next step. The other method is to utilize a form of parallelism known as pipelining. To implement pipelining, the resources must be divided into partitions and each partition assigned a single function. The results of each partition can then be passed directly into the next partition for further processing (12:239). Process speedup can be realized through pipelining when stages of the pipeline can be overlapped so that the various stages are being executed in parallel.

The implemented model uses a pipelined approach to perform a query of the form

$$\sigma R \bowtie \sigma S.$$

as shown in Figure 5.2. First, select operations are performed on the R and S input relations in the retrieval layer processors. The results of these select operations serve as input to an equi-join operation utilizing the bucket join algorithm. It assumes that both input relations are spread uniformly across the available disks and that there is one retrieval layer processor dedicated to each disk. The processing layer nodes are divided into two partitions with one dedicated to the hash processing step and the other dedicated to the

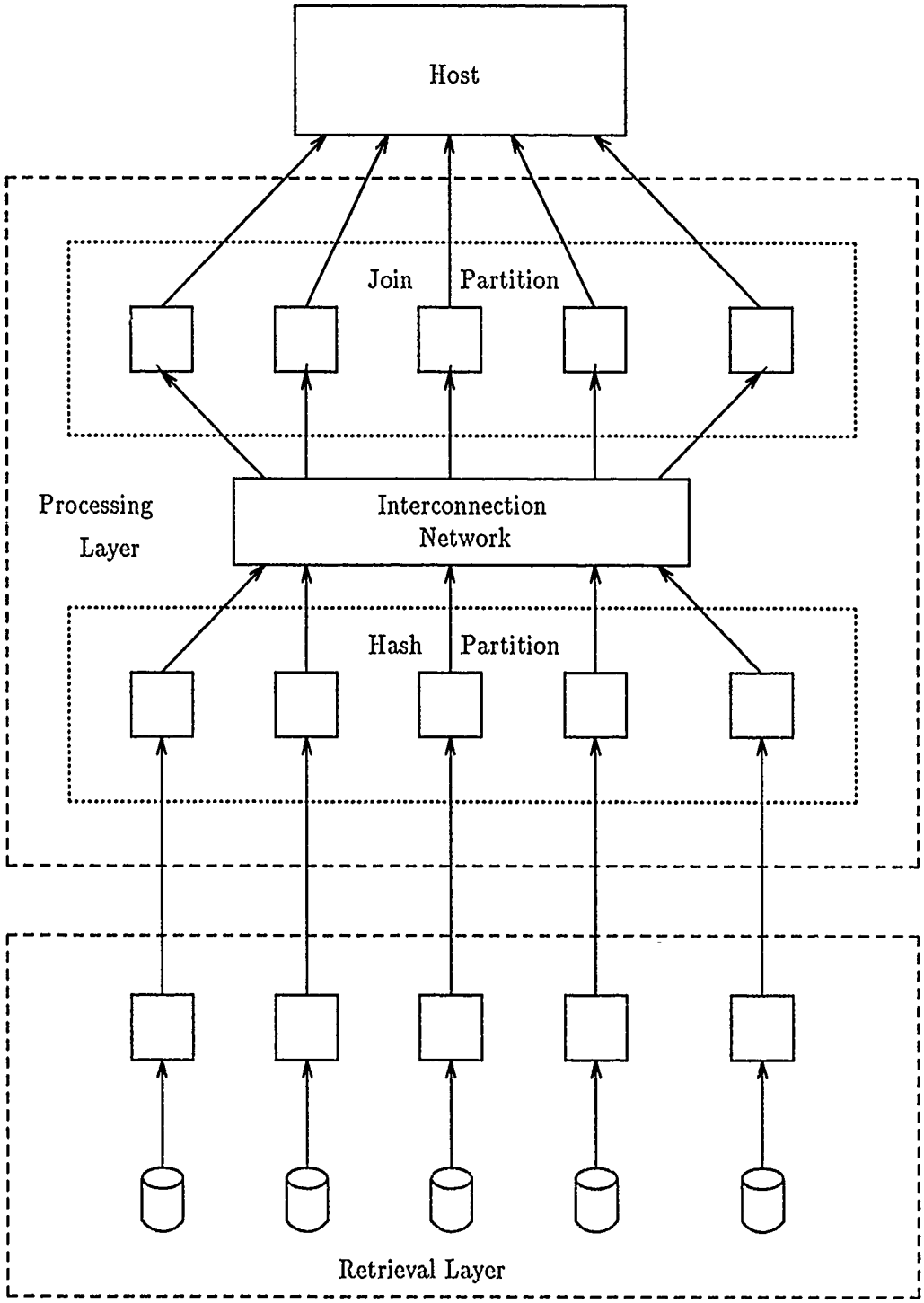
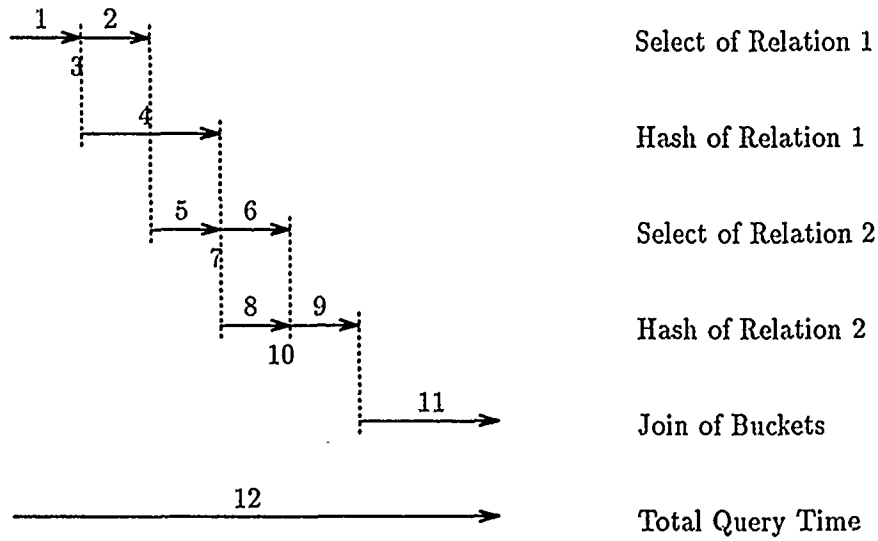


Figure 5.2. Architecture Data Flow for Multiple Step Query

join processing step using the local nested-loop algorithm. Utilizing this approach, all of the stages could be overlapped to some extent. However, this particular model assumes that only the select stage and the hash stage are overlapped. Figure 5.3 illustrates the stage overlapping that occurs for this implementation. Each of the arrows represents a particular operation in the query. The numbers on the arrows partition the operations into stages that may be overlapped. The extent to which the operations are overlapped is illustrated by the vertical dotted lines. The following sections provide a brief summary of the processing that occurs within each partition.

5.2.1 Select Partition All of the nodes in the select partition are part of the retrieval layer of the logical architecture. Each is connected to a unique disk node that contains a fragment of each of the input relations. Each node, therefore, performs a select operation on a fragment of each of the input relations. The select operations are performed in a consecutive manner such that the entire fragment of one relation is processed before the first block of the second relation fragment is read. The operation is performed by filling the memory of the processor with blocks of tuples from the input relation. Each block is then scanned with the select criteria being applied to each tuple. If a tuple meets the selection criteria, it is written into the result buffer which can hold one block of data. When the result buffer is filled, the block of data is immediately sent to the hash partition. This allows the hash processing to begin while the select operation is still being performed. This process continues until all of the blocks of the relation have been processed.

5.2.2 Hash Partition The nodes in the hash partition are part of the processing layer. Each of the hash nodes receives select result blocks from one particular retrieval layer node. Upon receiving a block of tuples, the hash process immediately begins applying the hash function to the tuples in the block. There is an output buffer, or bucket, associated with each processor assigned to the join partition. As each tuple is processed it is written to one of the output buffers. When an output buffer is filled, it is sent to the associated join node. As each finishes processing the second relation, it sends a hash complete message to all join nodes.



1 - Time to first block of output (T_{1sp})
2 - Time to complete select ($T_{sp} - T_{1sp}$)
3 - Time to complete entire select (T_{sp})
4 - Time to complete hash (T_h)
5 - Time to first block of output (T_{1sp})
6 - Time to complete select ($T_{sp} - T_{1sp}$)
7 - Time to complete entire select (T_{sp})
8 - Time of hash until last block received (T_{h2})
9 - Time of hash after last block received ($T_h - T_{h2}$)
10 - Time to complete hash (T_h)
11 - Time to join buckets (T_J)
12 - Time to perform entire query

Figure 5.3. Overlapping of Multiple Step Query

5.2.3 Join Partition The nodes in the join partition are part of the processing layer. As stated earlier, it is assumed for this implementation that all of the blocks for both of the relations to be joined will fit into the memory of the join processors. This eliminates the need for intermediate relation storage at this level. Each node receives blocks until it has received a hash complete message from each hash node. It then performs a nested-loop equi-join operation on the input relations. The result relation blocks produced by the join operation on each node are then sent to the cube manager to be merged into the final result relation.

5.3 Multiple Step Bucket Join Model Analysis

In order to model the multiple step query operations proposed by Kearns, two levels of interaction must be considered. First, the time required to perform each step of the operation must be represented in terms of the various system performance parameters. This level is modeled in a fashion similar to that of the single step models presented in Chapter IV. The second level of the model must consider the amount of function overlap that occurs when the operations are pipelined and the effect on total query time. The second level requires analyzing the problem from a higher level of abstraction. This section presents Kearns' models from the high level view. The low level models for each function are included in Appendix C.

A further examination of Figure 5.3 provides a description of the functions required to perform this query and the overlapping of the functions that occur. The first operation, the select on relation 1, can be broken into two parts. These are the time required to produce the first result block and the time to complete the rest of the select operation. This division is important because when the first result block is produced and sent to the hash node, the hash operation on the relation can begin. The time to produce the first result block is represented by T_{1sp}. The time to perform the entire select operation is represented by T_{sp}. The time to perform the hash function and send the produced buckets to the join nodes is represented by T_h. The third operation to be performed, the select operation on the relation 2, can begin as soon as the select operation on relation 1 is complete. The two parts of this second select operation can also be represented as T_{1sp}

and T_{sp}. The same overlap between the select operation and the hash operation can occur for relation 2 as for relation 1. For relation 2, however, the hash time must also be divided into two parts. They are the hash time until the point of receiving the last block to be hashed, T_{h2}, and the time required to complete the hash operation after the last block has been received, $T_h - T_{hs}$. This division is required because second part of the hash operation will not be overlapped with any other operations and the join step cannot begin until the hash of relation 2 is complete. The time to perform the join operation is represented as T_j. Given input relations of size R and S, where R_1 and S_1 represent the size of the select operation results and R_2 and S_2 represent the bucket sizes at each join node, total query time can then be expressed as:

$$T_{1sp}(R) + max \left| \begin{array}{c} (T_{sp}(R) - T_{1sp}(R)) + T_{1sp}(S) \\ or \\ T_h(R_1) \end{array} \right|$$

$$+max \left| \begin{array}{c} (T_{sp}(S) - T_{1sp}(S)) \\ or \\ T_{hs}(S_1) \end{array} \right| + (T_h(S_1) - T_{h2}(S_1)) + T_j(R_2, S_2).$$

5.4 Multiple Step Bucket Join Runtime Analysis

One limited test was run for the multiple step bucket join implementation. The purpose was to gain insight into the accuracy of this model for projecting total query time. For this test the select partition, the hash partition, and the join partition were each assigned 8 nodes. Each of the select nodes was assigned one unique disk node. The size of relation 1 was fixed at 240 blocks and the size of relation 2 was varied between 72 and 240 blocks. For all of the tests, both of the input relations were spread uniformly over the available disks. Figure 5.4 compares the predicted runtimes and the observed runtimes. The graph shows that the model predictions were very close to the observed runtimes over the entire test range. Over the test range, the average difference between the predicted and observed runtimes was about 3.5 %.

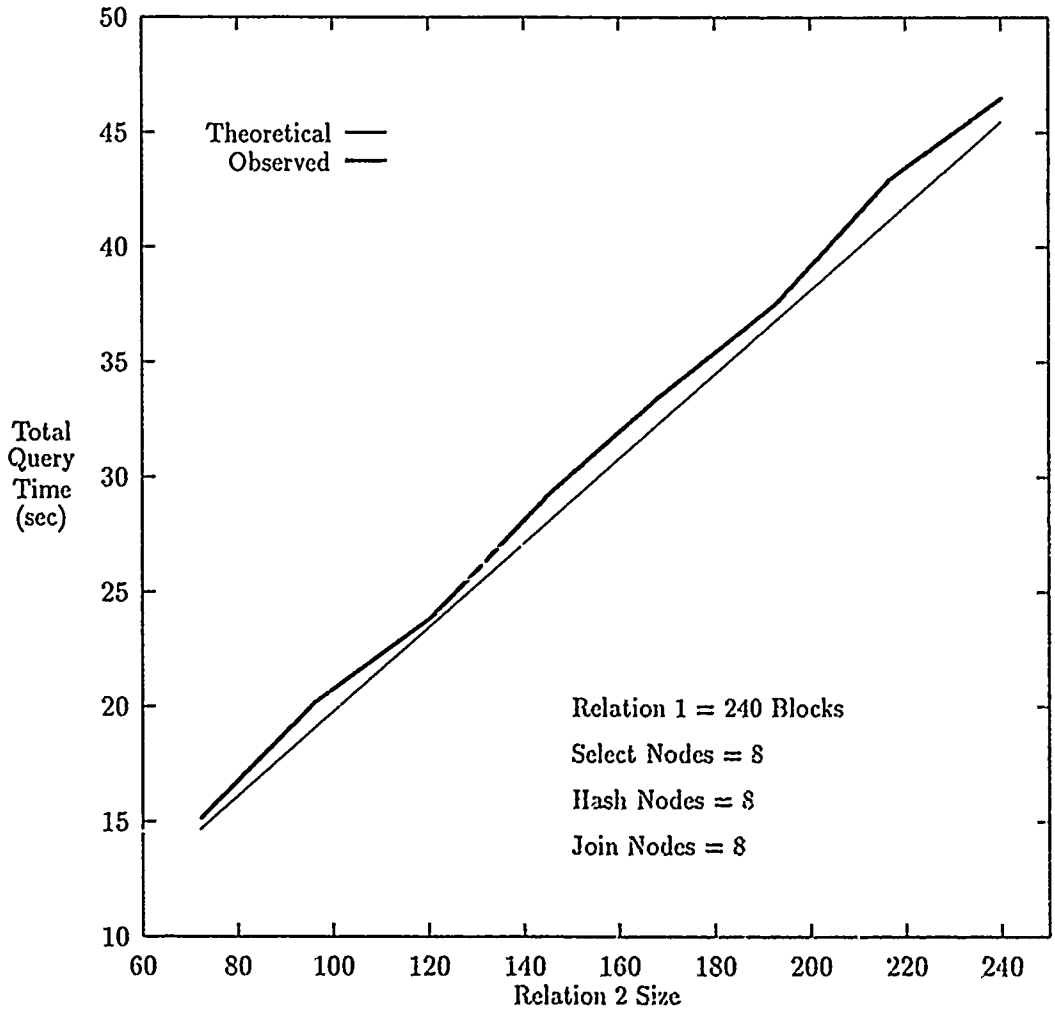


Figure 5.4. Multiple Step Query Results

VI. Conclusions and Recommendations

6.1 Conclusions

The purpose of this research was to show that relational database query performance can be improved utilizing Kearns' approach in the design and implemention of parallel database operations on a commercially available concurrent computer. Kearns developed over 200 analytical models to be used in the design of parallel implementations of a wide range of relational database operations. The models are categorized into the single step models, which model single operation queries, and multiple step models which model complex multiple operation queries. Due to the lack of time and resources, only a small subset of the models could actually be implemented. The single step implementation included the nested-loop join operation and the bucket join operation. A multiple step model was also implemented which consisted of two select operations feeding their results into a bucket join operation. The evaluation of the implementations involved a comparison of the theoretical performance predictions produced by Kearns' models with the observed performance on the iPSC/1.

The nested-loop join provided an example of a case in which the target architecture prohibited the implementation from matching exactly the associated model. The model assumed that the disks could efficiently broadcast a block of data to all of the join processors in some constant time. The iPSC/1 did not provide this capability. The communication between the disks and the join nodes was implemented in a ring structure instead. The implementation of the ring communication structure produced greater communication overhead than was predicted for the broadcast communication structure. As long as the number of join nodes used to implement the join operation was relatively small, the model was still fairly accurate. However, as the number of nodes was increased, the observed runtimes were much greater than the theoretical runtimes predicted by the model. Kearns' nested-loop model with broadcast communication was at least accurate to within about 8.5 % for all test cases run.

At this point, the model was altered to take into account the ring communication structure. The ring communication model was more accurate than the broadcast communi-

nication model. For all cases run, the ring communication model was at least accurate to within about 6 %. The evaluation of the nested-loop model shows that even when there are discrepancies between the model and the implementation, the model may be accurate under certain constrained conditions. However, if the limits of the model are not known by the user, inaccurate conclusions may be drawn. The successful alteration of the model to account for the ring communication structure also demonstrated that the model is flexible enough to be changed to account for major discrepancies between the model assumptions and the target architecture.

The single step bucket join model, on the other hand, was more accurate than the nested-loop model. For all of the test cases run, the bucket join model was at least accurate to within about 3 %. And for the cases with the larger input relation sizes, it was even more accurate. The analysis of the bucket join sensitivity tests showed that the model did a good job of predicting the impact of changing certain performance parameters. The parameters that were altered for the sensitiviy tests were the disk response time and the join selectivity factor. For the disk response time sensitivity test cases, the model predicted an average runtime increase of about 9 % when the response time was increased. The observed runtime increase was about 13 %. For the join selectivity factor sensitivity test cases, the model predicted an average runtime of about 13 % when the join selectivity factor was increased. The observed runtime increase was about 15 %. The evalnation of the bucket join model has demonstrated the accuracy of the model, and has also has demonstrated the usefulnes of the model for projecting the effect of altering various system performance parameters.

The tests run against the multiple step query implementation were very limited due to time constraints. The results that were produced, however, were promising. For the test cases that were run, the average difference between the theoretical and observed runtimes was about 3.5 %.

6.2 Recommendations

The results of this research show that for the cases demonstrated, Kearns' models are at least accurate to within about 8.5 %, with some being even more accurate under certain conditions. These results suggest that the models should prove useful in the design of par-

allel ralational database systems. Further work, however, is needed to further demonstrate the join models implemented in this thesis as well the other operations Kearns modeled. Some suggested areas of future research include the validation of the single step models when implemented for greater numbers of nodes, the evaluation of the models for cases in which the input relations are not uniformly distributed across all disks, the implementation and evaluation of the models on a different concurrent computer system, an indepth implementation and analysis of the multiple step models.

The number of nodes used to implement the single step models was limited to 16. The analysis of the nested-loop algorithm, however, showed that it is possible for the model performance predictions and the associated implementation's observed performance to be fairly consistent for small numbers of nodes but very inconsistent for larger numbers of nodes. It is essential, therefore, to implement and evaluate a modeled operation for larger numbers of nodes in order to make any statement about the the accuracy of the model as the number of nodes is increased.

One of Kearns' basic assumptions that carried over to this effort was that the input relations are always uniformly distributed across the available disks. In a real system, however, this will probably never be the case. Future efforts should investigate the possibility of modifying the models to handle cases in which this basic assumption is false. This may include the application of load balancing algorithms between the nodes to be run prior to the execution of the specified query.

The concept behind the development of a set of general models to use in the design of parallel database systems is to be able to vary the hardware perfomance parameters to make the models match a range of physical machines. This thesis has only addressed the validity of the approach when applied to one type of the machine, the iPSC/1 Hypercube. Future efforts should include implementations on other types of architecture.

The final area proposed for future research is an indepth investigation of the multiple step query models. The multiple step implementation for this thesis was very limited. It included one query with fixed partition sizes. A more indepth effort should investigate the impact of varying partition sizes on the total query time. For this implementation,

the partitions for all of the steps were all the same size. If the partition size for the join operation, which requires the greatest amount of processing, were made larger, it should drastically reduce the total query time.

Appendix A. *Single Step Models*

A.1 Model Performance Parameters

Kearns' single step query performance models are composed of a number of performance parameters that characterize the hardware the query is implemented on and the size and organization of the input data. This section defines each of these parameters and discusses their possible values.

T_c - T_c is the time required to compile a query. The implementation on the iPSC/1 does not compile queries from a generl query language. It offers the user specific pre-canned operations. Therefore for the tests run for this thesis, T_c was always equal to 0.

T_m - T_m is the time required to send a message between the backend and the host processor in milliseconds. Since the iPSC/1 implementation is not connected as a backend to a host processor, this number is always 0.

T_d - T_d is one of the disk performance parameters. It is the average disk access time in milliseconds. This includes the rotational latency and the initial seek to the desired track. This value is variable in the iPSC/1 implementation It can be set in the disk simulator routine that runs on the disk nodes.

p_b - p_b is the amount of processor memory in terms of input block size. This number can be specified in each node routine in the iPSC/1 implementation. When run, each node will limit itself to the amount of memory specified by this parameter.

p - p is number of processors used to perform the requested operation. This is a user input to the iPSC/1 implemenation and can be set to 4, 8, 12, or 16.

d - d is the number of disks that the input relations are spread across. For this thesis, the number of disks is always equal to the number of processors.

b - b is the number of blocks stored on one track on disk. This number can be specified in the disk simulator routine. For all of the tests run, this number was set at 10.

T_s - T_s is another one of the disk performance parameters. It is the time required for a track-to-track seek in milliseconds. This number can be specified in the disk simulator routine. For all of the tests run, this number was set to 10.

T_{io} - T_{io} is the final disk performance parameter. It is the amount of time required to transfer one block of data between a processor and a disk in milliseconds. It can also be set in the disk simulator routine.

R - R is the number of blocks of tuples in one of the input relations. This number is variable. However, to achieve a uniform distribution of blocks across the available disks, it should be a multiple of the number of disks.

S - S is the number of blocks of tuples in the second input relation. The same value guidelines apply to S as R.

T_{bt} - T_{bt} is the time to send one block of data between the backend processor and a node.

T_{bt} - T_{bt} is the time required to send one block of data to the backend processor. This translates as the time to send a block of data between the iPSC/1 host and a node in the cube. It was measured to be approximately 16 milliseconds.

T_{sc} - T_{sc} is the time required to scan one block of tuples from an input relation. This time was measured on the iPSC/1 for a blocksize of 100 tuples to be 7.5 milliseconds.

T_b - T_b is the time required to perform a join operation on one block of data from each relation. Utilizing the nested-loop join algorithm, this time was measured on the iPSC/1 to be 145 milliseconds.

jsf - jsf is the join selectivity factor. This is the size of the result relation expressed as a percentage of the cross product of the two input relations.

J_B - J_B is the number of blocks in the join result relation. It is calculated by the expression

$$[(jsf * ((R * (B/r)) * (S * (B/s)))) / (B/(r+s)))]$$

where B is the number of bytes per block, r is the R relation tuple size, and s is the S relation tuple size.

A.2 Nested-Loop Join

The performance model for the parallel nested-loop join algorithm is expressed as follows:

$$T_c + T_m + [T_d + (p_b) * T_s + p_b * T_{io}] * (R/(p_b * p))$$

$$+((R/p) * S * Tb) + [(S * (R/(p_b * p))) * T_{io}] + [(J_B/p) * T_{bt}]$$

This expression can be broken into five parts. The first part is the time required to compile the query and send the query from the host to the backend processor. It is expressed by

$$T_c + T_m$$

The next part is the total time that will be spent loading memory with blocks from the R relation. During the execution of the query, each block of the R, or smaller, relation will be read in to memory one time. This time is expressed as

$$[T_d + (p_b/p) * T_s + p_b * T_{io}] * (R/(p_b * p))$$

First the amount of time required to fill the memory one time is calculated by $[T_d + (p_b/p) * T_s + p_b * T_{io}]$. This expression is then multiplied by the number of times that each processor will have to fill its memory with blocks from the R relation, $(R/(p_b * p))$. Theoretically, the blocks will always perform this operation in parallel, therefore, the model only has to account for the memory fill time for one processor.

The next part of the expression is the time required to join a fragment from relation R of size R/p with all of the blocks from relation S. This time is expressed as

$$(R/p) * S * J_B$$

This calculates the number of blocks that must be joined from each relation and multiplies it by the time required to perform a join with one block of each relation. This join operation is where a majority of the run time for the nested-loop join operation is spent.

The next part of the expression is the communication time for the broadcast model. It is expressed as

$$[(S * (R/(p_b) * p))) * T_{io}]$$

First, the number of S relation blocks that must be transmitted is calculated by $(S * (R/(p_b) * p)))$. This is the size of the S relation multiplied by the number of R relation fragments that will be read in by each processor. Since the processors will broadcast their blocks of S in serial, and the model assumes that the time required to send a block of data to any other node is constant, this number is multiplied by the time required to send a block of data from one node to another.

The final part of the expression is the time required for the join processors to return the result tuples to the host processor. This time is expressed as

$$[(J_B/p) * T_{bt}$$

That is the number of results produced on each processor, (J_B/p), times the time require to send a block from a cube node to the host. Again, it is assumed that the processors will perform this communication in parallel.

A.3 Bucket Join

The performance model for the bucket join algorithm with the nested-loop is expressed as follows:

$$T_c + T_m + T_d + T_{io} + max \left| \begin{array}{c} (T_{sc} * (R/p)) \\ +2 * [((R/p) + 1) * (p - 1)] * Tbt \\ or \\ 2 * ((R/d) * Tio) + (((R/d)/b) * T_s) \\ +((R/p) + 1) * (T_d + T_{io}) \end{array} \right|$$

$$+T_d + Tio + max \left| \begin{array}{c} (T_{sc} * (S/p)) \\ +(2 * [((S/p)+1) * (p-1)] * Tbt) \\ or \\ 2 * ((S/d) * Tio) + (((S/d)/b) * T_s) \\ +((S/p)+1) * (T_d + T_{io}) \end{array} \right|$$
$$+[(T_d + (pb * T_{io})) * (((R/p)+1)/p_b)] + (((R/p)+1) * ((S/p)+1) * T_b)$$
$$(((S/p)+1) * (((R/p)+1)/p_b)) * T_{io}] + j_B * T_{bt} \qquad \text{(A.1)}$$

This expression can be broken into seven parts. As with the nested-loop model, the first part is the time required to compile the query and send the query from the host to the backend processor. It is expressed by

$$T_c + T_m$$

The next part is the time required to perform the hash function on the R relation. It is expressed as

$$T_d + T_{io} + max \left| \begin{array}{c} T_{sc} * (R/p)) \\ +(2 * [((R/p)+1) * (p-1)] * Tbt \\ or \\ 2 * ((R/d) * Tio) + (((R/d)/b) * T_s) \\ +((R/p)+1) * (T_d + T_{io}) \end{array} \right|$$

$T_d + T_{io}$ is the time required to find the proper track on disk to begin reading the R relation and the time required to transfer the first block to the processor. The max function then compares the time required to perform the actual scan of the data and send the bucket fragments to the proper nodes versus the time required to read the data from disk and then store the bucket fragments as they are received. This time is only calculated for one processor as the hash occurs in parallel across all of the processors.

The next part of the expression calculates the time required to perform the hash function on the S relation. It is expressed as

$$+T_d + Tio + max \left| \begin{array}{c} T_{sc} * (S/p)) \\ +(2 * [((S/p)+1) * (p-1)] * Tbt \\ or \\ 2 * ((S/d) * Tio) + (((S/d)/b) * T_s) \\ +((S/p)+1) * (T_d + T_{io}) \end{array} \right|$$

The S relation hash time is calculated in the same manner as the R relation hash time. The parts of the model expression discussed to this point express the hash step of the algorithm. The rest of the model expresses the time required to perform the join step.

The first part of the join step, expresses the time required to fill the processor memory with one fragment of blocks of the R relation. This time is expressed as

$$[[T_d + (p_b * T_{io})] * (((R/p)+1)/p_b)]$$

$T_d + (p_b * T_{io})$ expresses the time to fill the processor's memory one time. This time is multiplied by the number of times that the processor's memory will have to be filled which is expressed as $(((R/p)+1)/p_b)$.

The next part of the nested-loop step is the time required to perform the actual join processing. It is expressed as

$$(((R/p)+1) * ((S/p)+1) * T_b)$$

$((R/p)+1)$ is the size of one R relation fragment, it is multiplied by $((S/p)+1)$ which is the size of one S relation fragment. This multiplication results in the total number of block joins that must be performed. This number is then multiplied by T_b which the time required to perform one block join.

The next part of the expression,

$$[(((S/p) + 1) * (((R/p) + 1)/p_b)) * T_{io}]$$

represents the time required to read the blocks of the S relation fragment one at a time to be compared with the R relation blocks that have been loaded into memory.

The final part of the expression,

$$[(j_B * T_{bt}]$$

is the time required to send the result relation fragments to the backend processor.

Appendix B. *Multiple Step Low Level Models*

This appendix documents the low level multiple step query models. Two additional performance parameters are included in these models that were not listed in chapter IV. The first is selectivity factor, f, which is the percentage of the input relations that meet the criteria of a select operation. For the test cases run with these models, f is equal to .5. The second parameter is number of hash processors, p_h.

$\underline{T_{1sp}}$ - Time to produce and send first select result block to the appropriate hash processor.

$$T_{1sp} = T_m + T_d + T_{io} + max \left| \begin{array}{c} (r/(v*f))*T_{sc} \\ or \\ (((r/(v*f))/b)-1)*T_s \\ +(r/(v*f))*T_{io} \end{array} \right| + T_{bt}$$

$\underline{T_{sp}}$ - Total time for a select retreival operation.

$$T_{sp} = T_m + T_d + T_{io} + max \left| \begin{array}{c} (((R/p)-1)*T_{sc}) \\ +(((v*((R/p)*f)*(B/R)*(1/B))-1)*T_{bt} \\ or \\ (((R/p)/b)-1)*T_s + (R/p)*T_{io} \end{array} \right|$$

$$+T_{sc} + T_{bt}$$

$\underline{T_h}$ - Time to complete the hash function.

$$T_h = T_m + (R_1 * (T_{bt} + T_{sc})) + (((p_h - 1) * R_2) * T_{bt})$$

$\underline{T_{h2}}$ - Time of the hash function to the point of receiving the last block of data to be hashed.

$$T_{h2} = T_m + ((R_1 - 1) * (T_{bt} + T_{sc})) + (((p_h - 1) * (R_2) - 1) * T_{bt}$$

T_J - Time to perform the nested-loop join.

$$T_J = T_m + (R_2 * S_2 * T_b) + J_B * T_{bt}$$

R_1 - Number of result blocks produced by the select operation on relation R.

$$R_1 = ((v * (R * f) * (B/r) * (1/B)$$

S_1 - Number of result blocks produced by the select operation on relation S.

$$S_1 = ((v * (S * f) * (B/s) * (1/B)$$

R_2 - Number of blocks in each fragment of R to be joined.

$$R_2 = (R_1/(p_h - 1)) + 1$$

S_2 - Number of blocks in each fragment of S to be joined.

$$S_2 = (S_1/(p_h - 1)) + 1$$

Appendix C. *Test Data*

C.1 Test N-1

Test Description

The number of nodes used to perform the nested-loop join function was varied for fixed input relation sizes. The data tables contain entries for model 1, which is Kearns' original model, model 2, which is the version of the model which takes into to consideration the ring communication structure, and the observed runtimes from the iPSC/1.

Performance Parameters

$jsf = .00001, p_b = 14, T_d = 35, T_{io} = 20, T_s = 10$

Trial 1 ($R = 48, S = 48$)

Table C.1. Test N-1, Trial 1 Data

Nodes	Model 1 (sec)	Model 2 (sec)	Observed (sec)
4	84.655	85.212	84.704
8	42.334	43.092	44.256
12	28.226	29.052	31.688
16	21.173	22.032	24.844

Trial 2 ($R = 96, S = 96$)

Table C.2. Test N-1, Trial 2 Data

Nodes	Model 1 (sec)	Model 2 (sec)	Observed (sec)
4	338.021	340.250	337.725
8	169.017	169.711	169.915
12	112.682	113.785	114.000
16	84.514	85.821	90.620

Trial 3 ($R = 144$, $S = 144$)

Table C.3. Test N-1, Trial 3 Data

Nodes	Model 1 (sec)	Model 2 (sec)	Observed (sec)
4	760.110	765.124	751.207
8	380.061	383.378	378.460
12	253.378	254.209	252.460
16	190.036	191.380	193.440

C.2 Test N-2

Test Description

For the nested-loop join, the disk response time parameters, T_d and T_{io}, were varied while the number of processors was held constant at 8.

Trial 1

Performance Parameters

$p = 8, jsf = .00001, p_b = 14, R = 96, T_s = 10, T_d = 100, T_{io} = 20$

Table C.4. Test N-2, Trial 1 Data

S	Model (sec)	Observed (sec)
40	70.650	73.104
48	84.711	87.425
56	98.711	100.810
64	112.831	114.436
72	126.892	128.124
80	140.952	140.895
88	155.012	154.205
96	169.072	168.615

Trial 2

Performance Parameters

$p = 8, jsf = .00001, p_b = 14, R = 96, T_s = 10, T_d = 150, T_{io} = 40$

Table C.5. Test N-2, Trial 2 Data

S	Model (sec)	Observed (sec)
40	71.619	75.047
48	85.817	88.660
56	100.014	101.940
64	114.211	115.825
72	128.401	129.550
80	142.606	143.720
88	156.804	157.955
96	171.001	172.060

Trial 3

Performance Parameters

$p = 8, jsf = .00001, p_b = 14, R = 192, T_s = 10, T_d = 100, T_{io} = 20$

Table C.6. Test N-2, Trial 3 Data

S	Model (sec)	Observed (sec)
160	563.096	561.395
168	591.216	586.265
176	619.337	614.120
184	647.457	646.575
192	675.578	670.115

Trial 4

Performance Parameters

$p = 8, jsf = .00001, p_b = 14, R = 192, T_s = 10, T_d = 150, T_{io} = 40$

Table C.7. Test N-2, Trial 4 Data

S	Model (sec)	Observed (sec)
160	569.147	565.160
168	597.542	594.335
176	625.937	619.700
184	654.332	647.975
192	682.726	674.620

C.3 Test B-1

Test Description

The number of nodes used to perform the bucket join function was varied for fixed input relation sizes.

Performance Parameters

$jsf = .00001, p_b = 25, T_d = 35, T_{io} = 20, T_s = 10$

Trial 1 ($R = 144, S = 144$)

Table C.8. Test B-1, Trial 1 Data

Nodes	Model (sec)	Observed (sec)
4	207.702	201.155
8	56.811	54.865
12	27.510	26.265
16	16.795	16.555

Trial 2 ($R = 192, S = 192$)

Table C.9. Test B-1, Trial 2 Data

Nodes	Model (sec)	Observed (sec)
4	360.857	355.180
8	96.647	93.750
12	45.902	46.613
16	27.520	26.847

Trial 3 ($R = 240, S = 240$)

Table C.10. Test B-1, Trial 3 Data

Nodes	Model (sec)	Observed (sec)
4	556.040	548.153
8	147,000	142.070
12	68.973	67.485
16	40.894	38.960

C.4 Test B-2

Test Description

For the bucket join, the disk response time parameters, T_d and T_{io}, were varied while the number of processors was held constant at 8.

Trial 1

Performance Parameters

$p = 8, jsf = .00001, p_b = 25, R = 104, T_s = 10, T_d = 35, T_{io} = 20$

Table C.11. Test B-2, Trial 1 Data

S	Model (sec)	Observed (sec)
72	22.805	21.147
80	24.935	23.100
88	27.064	25.200
96	29.194	27.113
104	31.324	29.007

Trial 2

Performance Parameters

$p = 8, jsf = .00001, p_b = 25, R = 104, T_s = 10, T_d = 70, T_{io} = 40$

Table C.12. Test B-1, Trial 2 Data

S	Model (sec)	Observed (sec)
72	25.047	24.447
80	27.236	26.473
88	29.478	28.567
96	31.694	30.533
104	33.910	32.487

Trial 3

Performance Parameters

$p = 8, jsf = .00001, p_b = 25, R = 240, T_s = 10, T_d = 35, T_{io} = 20$

Table C.13. Test B-2, Trial 3 Data

S	Model (sec)	Observed (sec)
208	127.924	124.400
216	132.536	129.046
224	137.149	133.587
232	141.762	138.053
240	146.374	142.475

Trial 4

Performance Parameters

$p = 8, jsf = .00001, p_b = 25, R = 240, T_s = 10, T_d = 70, T_{io} = 40$

Table C.14. Test B-2, Trial 4 Data

S	Model (sec)	Observed (sec)
208	133.707	133.227
216	138.419	137.867
224	143.132	142.327
232	147.844	147.007
240	152.556	151.353

C.5 Test B-3

Test Description

For the bucket join, the join selectivity factor (jsf) was varied while the number of processors was held constant at 8. Tests were run with the jsf equal to .001 and the data was compared with the data from trials 1 and 3 from test B-2 where the jsf was set at .00001.

Trial 1

Performance Parameters

$p = 8, jsf = .001, p_b = 25, R = 104, T_s = 10, T_d = 35, T_{io} = 20$

Table C.15. Test B-3, Trial 1 Data

S	Model (sec)	Observed (sec)
72	25.770	24.093
80	28.230	26.960
88	30.689	28.727
96	33.148	32.513
104	35.607	33.980

Trial 2

Performance Parameters

$p = 8, jsf = .001, p_b = 25, R = 240, T_s = 10, T_d = 35, T_{io} = 20$

Table C.16. Test B-3, Trial 2 Data

S	Model (sec)	Observed (sec)
208	147.692	143.633
216	153,065	155.400
224	158,438	163.860
232	163.811	165.910
240	169.183	164.880

C.6 Test M-1

Test Description

For the multiple step join operation, the size of one of the relations was varied while the processor partition size were fixed (join nodes = hash nodes = select nodes = 8).

Performance Parameters

$jsf = .00001, p_b = 25, R = 240, T_s = 10, T_d = 35, T_{io} = 20$

Table C.17. Test M-1, Trial 1 Data

S	Model (sec)	Observed (sec)
72	14.669	15.153
96	19.070	20.187
120	23.470	23.813
144	27.870	29.053
168	32.270	33.453
192	36.670	37.440
216	41.070	42.867
240	45.471	46.500

Appendix D. *User's Guide*

D.1 Functional Overview

This software was developed to run specific parallel relational queries on generated input relations. It is not an operational relational database system. The functions that are currently implemented are the single step nested-loop join, the single step bucket join and one multiple step query which performs a select operation on two input relations and then performs a bucket join on the select result relations.

When the host program is run, it prompts the user for the function to be run, the numberof nodes to be used and the size of the input relations. The host program then loads the cube nodes with the proper programs to perform the requested operation and waits for the operation to complete. The total runtime for the function is then written to the screen.

D.2 File Summary

Following is a summary of the files required to run this software.

- host.c - When exectuted, host.c prompts the user for the desired query, the number of nodes to run the query on, and the sizes of the input relations. It then loads the cube nodes with the appropriate programs and initializes the query. It then receives the query results from the join nodes. Finally, it writes the total query time to the screen.
- disk.c - Disk.c is the program that is loaded and run on the disk nodes for all of the queries. It generates the input relations and then waits for I/O requests from the other nodes.
- nest.c - Nest.c is the program that is loaded and run on the join nodes for the single step nested-loop join query. It performs the nested-loop join utilizing the ring communication structure and passes the result relation blocks to the host.
- buck.c - Buck.c is the program that is loaded and run on the join nodes for the single step bucket join query. It performs both the hash processing step and the

join processing step for the bucket join algorithm and then passes the result relation blocks to the host.

- sel.c - Sel.c is the program that is loaded and run on the select nodes for the multiple step algorithm. It reads the input relations from the associated disk node, performs the select processing, and passes the result to the associated hash node.
- hash.c - Hash.c is the program that is loaded and run on the hash nodes for the multiple step algorithm. It receives the select result relation from the associated select node and performs the hash function on each tuple. It then passes the join buckets blocks to the appropriate join node.
- join.c - Join.c is the program that is loaded and run on the join nodes for the multiple step query. It receives blocks of tuples from for each relation from the hash nodes and performs a nested-loop join on the blocks. It passes the results to the host.

D.3 Setting the Parameters

There are a number of parameters that can be set by the user. Some of them are in the form of user input to system prompts at runtime, while others must be set in the code before compilation. The following sections discuss the parameters and how they are set. They also provide some guidance on the selection of the specific values.

D.3.1 Runtime Inputs

- num_nodes - This parameter specifies the number of nodes to be used to perform the single step queries. It can be set to 4, 8, 12, or 16. The number of nodes for the partitions of the multiple step query are hard coded.
- rela_size, relb_size - These parameters specify the number of blocks that are to be generated for the input relations. The blocks will be spread evenly across all of the available disk nodes. This parameter, therefore, MUST BE A MULTIPLE OF THE NUMBER OF DISK NODES!!

D.3.2 Pre-Compile Parameters

- BLOCK_SIZE - This parameter specifies the number of tuples contained in one input block. It is set in a define statement at the beginning of each of the files.
- ACCESS_TIME - This parameter specifies the time required to access the first block on disk for an I/O operation. It is set in a define statement at the beginning of the disk.c file.
- SEEK_TIME - This parameter specifies the time required to seek one track on disk. It is set in a define statement at the beginning of the disk.c file
- IO_TIME - This parameter specifies the time required to send one block of data from a disk node to a join node. It is set in a define statment at the beginning of the disk.c file.
- Join Selectivity Factor (jsf) - Join selectivity specifies the size of the join result relation as a percentage of the cross product of the two input relations. The jsf is set in the following statement found near the beginning of the main program of the disk.c file.

  num_joins = (((long)rela_size * BLOCK_SIZE)) * ((long)relb_size * BLOCK_SIZE)) /100000;

  The jsf for this statemnt is .00001. It is changed by altering the value of the denominator of the final division. (i.e. 10000 => .0001)
- MEM_SIZE - This parameter specifies the local memory size for the node where a file is running. It is set in a define statment at the beginning of each file.

D.4 Compilation Instructions

The following makefile can be used to compile the files needed to run the nested-loop join, the bucket join, and the multiple step query on the iPSC/1.

```
#
# Makefile Parallel Relational Queries on iPSC/1.
#
#
CFLAGS= -Alhu -K


app   : host nest buck sel disk hash join

host: host.c
cc -Alfu -o host host.c -lx  /usr/ipsc/lib/chost.a

nest: nest.o
ld -Ml -o nest /usr/ipsc/lib/Lcrtn0.o \
    nest.o \
   /usr/ipsc/lib/Llibcnode.a

buck: buck.o
ld -Ml -o buck /usr/ipsc/lib/Lcrtn0.o \
    buck.o \
   /usr/ipsc/lib/Llibcnode.a

sel: sel.o
       ld -Mh -o sek /usr/ipsc/lib/Lcrtn0.o\
       sel.o \
       /usr/ipsc/lib/Libcnode.a

disk: disk.o
ld -Mh -o disk /usr/ipsc/lib/Lcrtn0.o \
    disk.o \
   /usr/ipsc/lib/Llibcnode.a

hash: hash.o
ld -Ml -o hash /usr/ipsc/lib/Lcrtn0.o \
    hash.o \
   /usr/ipsc/lib/Llibcnode.a

join: join.o
ld -Mh -o join /usr/ipsc/lib/Lcrtn0.o \
    join.o \
   /usr/ipsc/lib/Llibcnode.a

clean :
rm  *.o
```

Appendix E. *Data Generation*

The data generation routine produces the join field values for the input relations based upon a specified join selectivity factor (jsf). The jsf is the size of the join result relation expressed as a percentage of the cross product of the two input relations. For example, if the specified jsf equals .00001, and the input relations each contain 100 blocks with 100 tuples in each, the cross product relation will have $10000 * 10000$ tuples. The join result relation, therefore, should have $(10000 * 10000) * .00001$, or 1000, tuples.

The jsf for the result relation produced by the join of two input relations generated by the data generation routine, will be approximately equal to that specified by the user. This is illustrated in Table E.1. This table shows the true selectivity factors that resulted from the join of input relations of varying sizes with the specified jsf equal to .001. The table lists the input relation sizes, the theoretical size of the result relation (in tuples), the observed result relation size, and the actual jsf.

Table E.1. Result Relation Sizes (jsf = .001)

Relation Sizes	Theoretical Result Tuples	Actual Result Tuples	Actual jsf
56 x 104	58240	60552	.00104
64 x 104	66560	74108	.00111
72 x 104	74880	79801	.00111
80 x 104	83200	96126	.00116
88 x 104	91520	95152	.00104
96 x 104	99840	113282	.00113
104 x 104	108160	111629	.00103

Bibliography

1. Banerjee, Jayanta, et al. "DBC-A Database Computer for Very Large Databases." In Hurson, A. R., et al., editors, *Tutorial: Parallel Architectures for Database Systems*, Washington D. C.: IEEE Computer Society Press, 1989.
2. Boral, Haran and David J. Dewitt. "Applying Data Flow Techniques to Data Base Machines." In Hurson, A. R., et al., editors, *Tutorial: Parallel Architectures for Database Systems*, Washington D. C.: IEEE Computer Society Press, 1989.
3. Boral, Haran and David J. Dewitt. "Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines." In Hurson, A. R., et al., editors, *Tutorial: Parallel Architectures for Database Systems*, Washington D. C.: IEEE Computer Society Press, 1989.
4. Cloud, K., et al. *Parallel Joins on the Mark III Hypercube*. Technical Report, Pasadena, CA: Jet Propulsion Laboratory, California Institute of Technology, 1989.
5. Dewitt, David J. "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems." In Hurson, A. R., et al., editors, *Tutorial: Parallel Architectures for Database Systems*, Washington D. C.: IEEE Computer Society Press, 1989.
6. Fox, Geoffrey C., et al. *Solving Problems On Concurrent Processors*, Volume I. Englewood Cliffs, New Jersey 07632: Prentice Hall, 1988.
7. Gardarin, Georges, et al. "SABRE: A Relational Database System for a Multimicroprocessor Machine." In Hsiao, David K., editor, *Advanced Database Machine Architecture*, Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1983.
8. Hawthorn, Paula B. and David J. DeWitt. "Performance Analysis of Alternative Database Machine Architectures," *IEEE Transactions of Software Engineering*, *SE-8*(1):61--75 (January 1982).
9. Hsiao, David K., et al. "The Implementation of a Multibackend Database System (MDBS): Part I-An Exercise in Database Software Engineering." In Hsiao, David K., editor, *Advanced Database Machine Architecture*, Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1983.
10. Hurson, A.R., et al., editors. *Tutorial: Parallel Architectures for Database Systems*. Washington D. C.: IEEE Computer Society Press, 1989.
11. Intel Corporation. *iPSC System Overview Manual*, Nov 1986.
12. Kearns, Timothy G. *A Methodology, Based on Analytical Modeling, For the Design of Parallel and Distributed Architectures for Relational Database Query Processors*. PhD dissertation, School of Engineering, Air Force Institute of Technology (AU), Wright Patterson AFB, OH, 1987 (AD-A189 844).
13. Korth, Henry F. and Abraham Silberschatz. *Database System Concepts*. McGraw-Hill Book Company, 1986.

14. Lakshmi, Seetha M. and Philip S. Yu. "Limiting Factors of Join Performance on Parallel Processors." In *Proceedings of the IEEE Fifth International Conference on Data Engineering*, Washington D. C.: IEEE Computer Society Press, 1989. ESBN 0-8186-1915-5.

15. Ozkarahan, Esen. *Database Machines and Database Management*. Englewood Cliffs, New Jersey: Prentice-Hall, inc., 1986.

16. Schuster, Stewart A., et al. "RAP.2-An Associative Processor for Databases and Its Applications." In Hurson, A. R., et al., editors, *Tutorial: Parallel Architectures for Database Systems*, Washington D. C.: IEEE Computer Society Press, 1989.

17. Strawser, Paula R. *A Methodology for Benchmarking Relational Database Machines*. PhD dissertation, Naval Post Graduate School, Monterey, CA, 1984.

18. Ullman, Jeffrey D. *Principles of Database Systems* (2 Edition). Computer Science Press, 1982.

19. Xin-Gui, He, et al. "The Implementation of a Multibackend Database System (MDBS): Part II-The Design of a Prototype MDBS." In Hsiao, David K., editor, *Advanced Database Machine Architecture*, Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1983.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE

AN EMPIRICAL EVALUATION OF ANALYTICAL MODELS FOR PARALLEL RELATIONAL DATABASE QUERIES

5. FUNDING NUMBERS

6. AUTHOR(S)

Mark C. Denham, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCs/ENG/90D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper documents the design and implementation of three parallel join algorithms to be used in the verification of analytical models developed by Kearns. Kearns developed a set of analytical models for a variety of relational database queries. These models serve as tools for the design of parallel relational database systems. Each of Kearns' models is classified as either single step or multiple step. The single step models reflect queries that require only one operation while the multiple step models reflect queries that require multiple operations. Three parallel join algorithms were implemented based upon Kearns' models. Two are based upon single step join models and one is based upon a multiple step join model. They are implemented on an Intel iPSC/1 parallel computer. The single step join algorithms include the parallel nested-loop join and the bucket (or hash) join. The multiple step algorithm that was implemented is a pipelined version of the bucket join. The results show that within the constraints of the test cases run, the three models are all at least accurate to within about 8.5 % and they should prove useful in the design of parallel relational database systems.

14. SUBJECT TERMS

Parallel Processing, Data Bases, Distributed Data Processing, Hypercube

15. NUMBER OF PAGES

115

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of . ; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

★U.S.GPO 1990 0-273-271